An Introduction to Software Architecture

David Garlan and Mary Shaw
January 1994

CMU-CS-94-166

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Also published as "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering, Volume I*, edited by V.Ambriola and G.Tortora, World Scientific Publishing Company, New Jersey, 1993.

Also appears as CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21.

This work was funded in part by the Department of Defense Advanced Research Project Agency under grant MDA972-92-J-1002, by National Science Foundation Grants CCR-9109469 and CCR-9112880, and by a grant from Siemens Corporate Research. It was also funded in part by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government, the Department of Defense, the National Science Foundation, Siemens Corporation, or Carnegie Mellon University.

Keywords: Software architecture, software design, software engineering

Abstract

As the size of software systems increases, the algorithms and data structures of the computation no longer constitute the major design problems. When systems are constructed from many components, the organization of the overall system—the software architecture—presents a new set of design problems. This level of design has been addressed in a number of ways including informal diagrams and descriptive terms, module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms.

In this paper we provide an introduction to the emerging field of software architecture. We begin by considering a number of common architectural styles upon which many systems are currently based and show how different styles can be combined in a single design. Then we present six case studies to illustrate how architectural representations can improve our understanding of complex software systems. Finally, we survey some of the outstanding problems in the field, and consider a few of the promising research directions.

Contents

List of Figures

1. Introduction

As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

This is the software architecture level of design. There is a considerable body of work on this topic, including module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms. In addition, an implicit body of work exists in the form of descriptive terms used informally to describe systems. And while there is not currently a well-defined terminology or notation to characterize architectural structures, good software engineers make common use of architectural principles when designing complex software. Many of the principles represent rules of thumb or idiomatic patterns that have emerged informally over time. Others are more carefully documented as industry and scientific standards.

It is increasingly clear that effective software engineering requires facility in architectural software design. First, it is important to be able to recognize common paradigms so that high-level relationships among systems can be understood and so that new systems can be built as variations on old systems. Second, getting the right architecture is often crucial to the success of a software system design; the wrong one can lead to disastrous results. Third, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives. Fourth, an architectural system representation is often essential to the analysis and description of the high-level properties of a complex system.

In this paper we provide an introduction to the field of software architecture. The purpose is to illustrate the current state of the discipline and examine the ways in which architectural design can impact software design. The material presented here is selected from a semester course, Architectures for Software Systems, taught at CMU by the authors [1]. Naturally, a short paper such as this can only briefly highlight the main features of the terrain. This selection emphasizes informal descriptions omitting much of the course's material on specification, evaluation, and selection among design alternatives. We hope, nonetheless, that this will serve to illuminate the nature and significance of this emerging field.

In the following section we outline a number of common architectural styles upon which many systems are currently based, and show how heterogeneous styles can be combined in a single design. Next we use six case studies to illustrate how architectural representations of a software system can improve our understanding of complex systems. Finally, we survey some of the outstanding problems in the field, and consider a few of the promising research directions.

The text that makes up the bulk of this article has been drawn from numerous other publications by the authors. The taxonomy of architectural styles and the case studies have incorporated parts of several published papers [1, 2, 3, 4]. To a lesser extent material has been drawn from other articles by the authors [5, 6, 7].

2. From Programming Languages to Software Architecture

One characterization of progress in programming languages and tools has been regular increases in abstraction level—or the conceptual size of software designers building blocks. To place the field of Software Architecture into perspective let us begin by looking at the historical development of abstraction techniques in computer science.

2.1. High-level Programming Languages

When digital computers emerged in the 1950s, software was written in machine language; programmers placed instructions and data individually and explicitly in the computer's memory. Insertion of a new instruction in a program might require hand-checking of the entire program to update references to data and instructions that moved as a result of the insertion. Eventually it was recognized that the memory layout and update of references could be automated, and also that symbolic names could be used for operation codes, and memory addresses. Symbolic assemblers were the result. They were soon followed by macro processors, which allowed a single symbol to stand for a commonly-used sequence of instructions. The substitution of simple symbols for machine operation codes, machine addresses yet to be defined, and sequences of instructions was perhaps the earliest form of abstraction in software.

In the latter part of the 1950s, it became clear that certain patterns of execution were commonly useful—indeed, they were so well understood that it was possible to create them automatically from a notation more like mathematics than machine language. The first of these patterns were for evaluation of arithmetic expressions, for procedure invocation, and for loops and conditional statements. These insights were captured in a series of early high-level languages, of which Fortran was the main survivor.

Higher-level languages allowed more sophisticated programs to be developed, and patterns in the use of data emerged. Whereas in Fortran data types served primarily as cues for selecting the proper machine instructions, data types in Algol and it successors serve to state the programmer's intentions about how data should be used. The compilers for these languages could build on experience with Fortran and tackle more sophisticated compilation problems. Among other things, they checked adherence to these intentions, thereby providing incentives for the programmers to use the type mechanism.

Progress in language design continued with the introduction of modules to provide protection for related procedures and data structures, with the separation of a module's specification from its implementation, and with the introduction of abstract data types.

2.2. *Abstract Data Types*

In the late 1960s, good programmers shared an intuition about software development: If you get the data structures right, the effort will make development of the rest of the program much easier. The abstract data type work of the 1970s can be viewed as a development effort that converted this intuition into a real theory. The conversion from an intuition to a theory involved understanding

- *the software structure* (which included a representation packaged with its primitive operators),
- *specifications* (mathematically expressed as abstract models or algebraic axioms),
- *language issues* (modules, scope, user-defined types),
- *integrity of the result* (invariants of data structures and protection from other manipulation),
- *rules for combining types* (declarations),
- *information hiding* (protection of properties not explicitly included in specifications).

The effect of this work was to raise the design level of certain elements of software systems, namely abstract data types, above the level of programming language statements or individual algorithms. This form of abstraction led to an understanding of a good organization for an entire module that serves one particular purpose. This involved combining representations, algorithms, specifications, and functional interfaces in uniform ways. Certain support was required from the programming language, of course, but the abstract data type paradigm allowed some parts of systems to be developed from a vocabulary of data types rather than from a vocabulary of programming-language constructs.

2.3. *Software Architecture*

Just as good programmers recognized useful data structures in the late 1960s, good software system designers now recognize useful system organizations.

One of these is based on the theory of abstract data types. But this is not the only way to organize a software system.

Many other organizations have developed informally over time, and are now part of the vocabulary of software system designers. For example, typical descriptions of software architectures include synopses such as (italics ours):

- "Camelot is based on the *client-server model* and uses remote procedure calls both locally and remotely to provide communication among applications and servers."[8]
- "*Abstraction layering* and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a *client-server model* for the structuring of applications."[9]
- "We have chosen a *distributed, object-oriented approach* to managing information." [10]
- "The easiest way to make the canonical sequential compiler into a concurrent compiler is to *pipeline* the execution of the compiler phases over a number of processors. . . . A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program."[11]

Other software architectures are carefully documented and often widely disseminated. Examples include the International Standard Organization's Open Systems Interconnection Reference Model (a layered network architecture) [12], the NIST/ECMA Reference Model (a generic software engineering environment architecture based on layered communication substrates) [13, 14], and the X Window System (a distributed windowed user interface architecture based on event triggering and callbacks) [15].

We are still far from having a well-accepted taxonomy of such architectural paradigms, let alone a fully-developed theory of software architecture. But we can now clearly identify a number of architectural patterns, or styles, that currently form the basic repertoire of a software architect.

3. Common Architectural Styles

We now examine some of these representative, broadly-used architectural styles. Our purpose is to illustrate the rich space of architectural choices, and indicate what are some of the tradeoffs in choosing one style over another.

To make sense of the differences between styles, it helps to have a common framework from which to view them. The framework we will adopt is to treat an architecture of a specific system as a collection of computational components—or simply *components*-—together with a description of the interactions between these components—the *connectors*. Graphically speaking, this leads to a view of an abstract architectural description as a graph in which the nodes represent the components and the arcs represent the connectors. As we will see, connectors can represent interactions as varied as procedure call, event broadcast, database queries, and pipes.

An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of *constraints* on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition.

Given this framework, we can understand what a style is by answering the following questions: What is the structural pattern—the components, connectors, and constraints? What is the underlying computational model? What are the essential invariants of the style? What are some common examples of its use? What are the advantages and disadvantages of using that style? What are some of the common specializations?

3.1. Pipes and Filters

In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. Hence components are termed “filters”. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”.

Among the important invariants of the style, filters must be independent entities: in particular, they should not share state with other filters. Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output of a pipe and filter network should not depend on the order in which the filters perform their incremental processing—although fair scheduling can be assumed. (See [5] for an in-depth discussion of this style and its formal properties.) Figure 1 illustrates this style.

Common specializations of this style include *pipelines*, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters have a well-defined type.

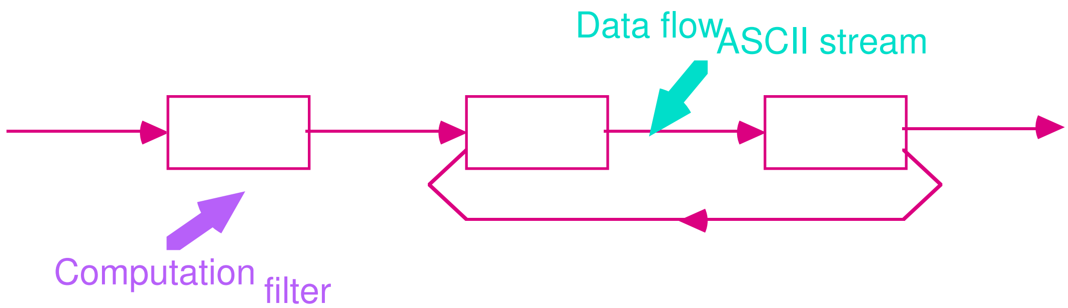

Figure 1: Pipes and Filters

A degenerate case of a pipeline architecture occurs when each filter processes all of its input data as a single entity.[1] In this case the architecture becomes a "batch sequential" system. In these systems pipes no longer serve the function of providing a stream of data, and therefore are largely vestigial. Hence such systems are best treated as instances of a separate architectural style.

The best known examples of pipe and filter architectures are programs written in the Unix shell [16]. Unix supports this style by providing a notation for connecting components (represented as Unix processes) and by providing run time mechanisms for implementing pipes. As another well-known example, traditionally compilers have been viewed as a pipeline systems (though the phases are often not incremental). The stages in the pipeline include lexical analysis, parsing, semantic analysis, code generation. (We return to this example in the case studies.) Other examples of pipes and filters occur in signal processing domains [17], functional programming [18], and distributed systems [19].

Pipe and filter systems have a number of nice properties. First, they allow the designer to understand the overall input/output behavior of a system as a simple composition of the behaviors of the individual filters. Second, they support reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them. Third, systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones. Fourth, they permit certain kinds of specialized analysis, such as throughput and deadlock analysis. Finally, they naturally support concurrent execution. Each filter can be implemented as a separate task and potentially executed in parallel with other filters.

But these systems also have their disadvantages.[2] First, pipe and filter systems often lead to a batch organization of processing. Although filters can

---

[1] In general, we find that the boundaries of styles can overlap. This should not deter us from identifying the main features of a style with its central examples of use.

[2] This is true in spite of the fact that pipes and filters, like every style, has a set of devout religious followers—people who believe that all problems worth solving can best be solved using that particular style.

process data incrementally, since filters are inherently independent, the designer is forced to think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character, pipe and filter systems are typically not good at handling interactive applications. This problem is most severe when incremental display updates are required, because the output pattern for incremental updates is radically different from the pattern for filter output. Second, they may be hampered by having to maintain correspondences between two separate, but related streams. Third, depending on the implementation, they may force a lowest com-

mon denominator on data transmission, resulting in added work for each filter to parse and unparse its data. This, in turn, can lead both to loss of performance and to increased complexity in writing the filters themselves.

3.2. Data Abstraction and Object-Oriented Organization

In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects—or, if you will, instances of the abstract data types. Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and (b) that the representation is hidden from other objects. Figure 2 illustrates this style.[3]

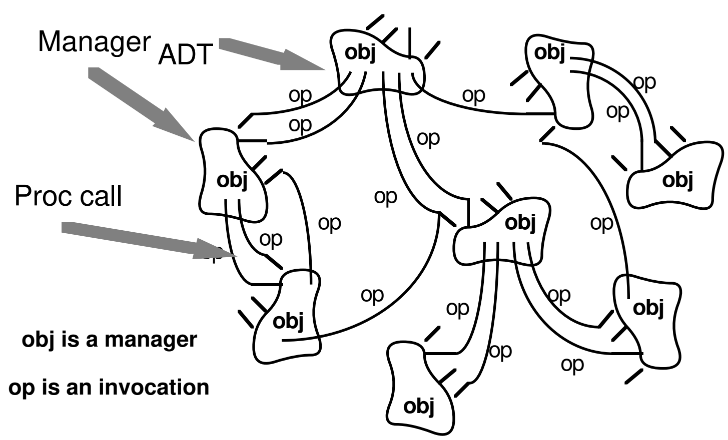

Figure 2: Abstract Data Types and Objects

[3]We haven't mentioned inheritance in this description. While inheritance is an important organizing principle for defining the types of objects in a system, it does not have a direct architectural function. In particular, in our view, an inheritance relationship is not a connector, since it does not define the interaction between components in a system. Also, in an architectural setting inheritance of properities is not restricted to object types—but may include connectors and even architectural styles.

The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread. There are many variations. For example, some systems allow "objects" to be concurrent tasks; others allow objects to have multiple interfaces [20, 21].

Object-oriented systems have many nice properties, most of which are well known. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other object. This is in contrast, for example, to pipe and filter systems, where filters do need not know what other filters are in the system in order to interact with them. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. In a module-oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module. Further there can be side-effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa.

3.3. Event-based, Implicit Invocation

Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines. However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast. This style has historical roots in systems based on actors [22], constraint satisfaction, daemons, and packet-switched networks.

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement ``implicitly'' causes the invocation of procedures in other modules.

For example, in the Field system [23], tools such as editors and variable monitors register for a debugger's breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke methods in those registered tools. These methods might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools (if any) are concerned with that event, or what they will do when that event is announced.

Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events. Procedures may be called in the usual way. But in addition, a component can register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time. Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls.

The main invariant of this style is that announcers of events do not know which components will be affected by those events. Thus components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events. For this reason, most implicit invocation systems also include explicit invocation (i.e., normal procedure call) as a complementary form of interaction.

Examples of systems with implicit invocation mechanisms abound [7]. They are used in programming environments to integrate tools [23, 24], in database management systems to ensure consistency constraints [22, 25], in user interfaces to separate presentation of data from applications that manage the data [26, 27], and by syntax-directed editors to support incremental semantic checking [28, 29].

One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system. A second benefit is that implicit invocation eases system evolution [30]. Components may be replaced by other components without affecting the interfaces of other components in the system.

In contrast, in a system based on explicit invocation, whenever the identity of a that provides some system function is changed, all other modules that import that module must also be changed.

The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it has no idea what other components will respond to it. Worse, even if it does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked. Nor can it know when they are finished. Another problem concerns exchange of data. Sometimes data can be passed with the event. But in other situations event systems must rely on a shared repository for interaction. In these cases global performance and resource management can become a serious issue. Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked. This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about an invocation of it.

3.4. Layered Systems

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers. Figure 3 illustrates this style.

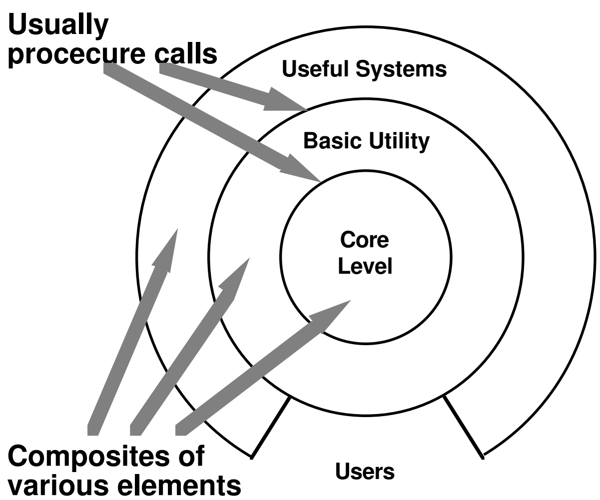

Figure 3: Layered Systems

The most widely known examples of this kind of architectural style are layered communication protocols [31]. In this application area each layer provides a substrate for communication at some level of abstraction. Lower levels define lower levels of interaction, the lowest typically being defined by hardware connections. Other appli-cation areas for this style include database systems and operating systems [9, 32, 33].

Layered systems have several desirable properties. First, they support design based on increasing levels of abstraction. This allows implementors to partition a complex problem into a sequence of incremental steps. Second, they support enhancement. Like pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers. Third, they support reuse. Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers. This leads to the possibility of defining standard layer interfaces to which different implementors can build. (A good example is the OSI ISO model and some of the X Window System protocols.)

But layered systems also have disadvantages. Not all systems are easily structured in a layered fashion. (We will see an example of this later in the case studies.) And even if a system *can* logically be structured as layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations. Additionally, it can be quite difficult to find the right levels of abstraction. This is particularly true for standardized layered models. One notes that the communications community has had some difficulty mapping existing protocols into the ISO framework: many of those protocols bridge several layers.

In one sense this is similar to the benefits of implementation hiding found in abstract data types. However, here there are multiple levels of abstraction and implementation. They are also similar to pipelines, in that components communicate at most with one other component on either side. But instead of simple pipe read/write protocol of pipes, layered systems can provide much richer forms of interaction. This makes it difficult to define system-independent layers (as with filters)—since a layer must support the specific protocols at its upper and lower boundaries. But it also allows much closer interaction between layers, and permits two-way transmission of information.

3.5. Repositories

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems.

The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard.

Figure 4 illustrates a simple view of a blackboard architecture. (We will examine more detailed models in the case studies.) The blackboard model is usually presented with three major parts:

The knowledge sources: separate, independent parcels of application-dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.

The blackboard data structure: problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

Control: driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

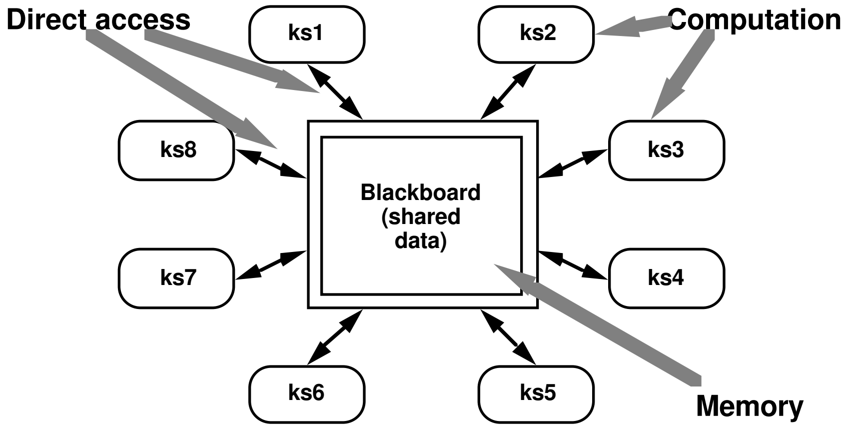

Figure 4: The Blackboard

In the diagram there is no explicit representation of the control component. Invocation of a knowledge source is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these.

Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition. Several of these are surveyed by Nii [34]. They have also appeared in other kinds of systems that involve shared access to data with loosely coupled agents [35].

There are, of course, many other examples of repository systems. Batch-sequential systems with global databases are a special case. Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments [36]. Even applications that have been traditionally viewed as pipeline architectures, may be more accurately interpreted as repository systems. For example, as we will see later, while a compiler architecture has traditionally been presented as a pipeline, the "phases" of most modern compilers operate on a base of shared information (symbol tables, abstract syntax tree, etc.).

3.6. Table Driven Interpreters

In an interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of *its* execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated. (See Figure 5.)

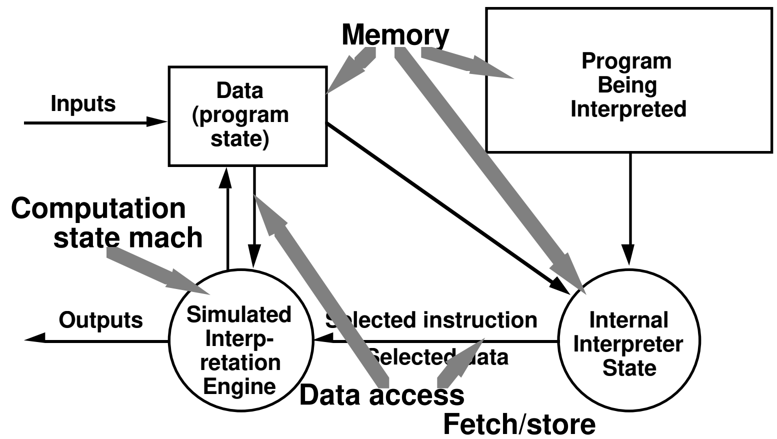

Figure 5: Interpreter

Interpreters are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. We occasionally speak of a programming language as providing, say, a "virtual Pascal machine."

We will return to interpreters in more detail in the case studies.

3.7. *Other Familiar Architectures*

There are numerous other architectural styles and patterns. Some are widespread and others are specific to particular domains. While a complete treatment of these is beyond the scope of this paper, we briefly note a few of the important categories.

- **Distributed processes:** Distributed systems have developed a number of common organizations for multi-process systems [37]. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in terms of the kinds of inter-process protocols that are used for communication (e.g., heartbeat algorithms).

One common form of distributed system architecture is a "client-server" organization [38]. In these systems a server represents a process that provides services to other processes (the clients). Usually the server does not know in advance the identities or number of clients that will access it at run time. On the other hand, clients know the identity of a server (or can find it out through some other server) and access it by remote procedure call.

- **Main program/subroutine organizations:** The primary organization of many systems mirrors the programming language in which the system is written. For languages without support for modularization this often results in a system organized around a main program and a set of subroutines. The main program acts as the driver for the subroutines, typically providing a control loop for sequencing through the subroutines in some order.

- **Domain-specific software architectures:** Recently there has been considerable interest in developing "reference" architectures for specific domains [39]. These architectures provide an organizational structure tailored to a family of applications, such as avionics, command and control, or vehicle management systems. By specializing the architecture to the domain, it is possible to increase the descriptive power of structures. Indeed, in many cases the architecture is sufficiently constrained that an executable system can be generated automatically or semi-automatically from the architectural description itself.

- **State transition systems:** A common organization for many reactive systems is the state transition system [40]. These systems are defined in terms a set of states and a set of named transitions that move a system from one state to another.

- **Process control systems:** Systems intended to provide dynamic control of a physical environment are often organized as process control systems [41]. These systems are roughly characterized as a feedback loop in which inputs from sensors are used by the process control system to determine a set of outputs that will produce a new state of the environment.

3.8. Heterogeneous Architectures

Thus far we have been speaking primarily of "pure" architectural styles. While it is important to understand the individual nature of each of these styles, most systems typically involve some combination of several styles.

There are different ways in which architectural styles can be combined. One way is through hierarchy. A component of a system organized in one architectural style may have an internal structure that is developed a completely different style. For example, in a Unix pipeline the individual components may be represented internally using virtually any style—including, of course, another pipe and filter, system.

What is perhaps more surprising is that connectors, too, can often be hierarchically decomposed. For example, a pipe connector may be implemented internally as a FIFO queue accessed by insert and remove operations.

A second way for styles to be combined is to permit a single component to use a mixture of architectural connectors. For example, a component might access a repository through part of its interface, but interact through pipes with other components in a system, and accept control information through another part of its interface. (In fact, Unix pipe and filter systems do this, the file system playing the role of the repository and initialization switches playing the role of control.)

Another example is an “active database”. This is a repository which activates external components through implicit invocation. In this organization external components register interest in portions of the database. The database automatically invokes the appropriate tools based on this association. (Blackboards are often constructed this way; knowledge sources are associated with specific kinds of data, and are activated whenever that kind of data is modified.)

A third way for styles to be combined is to completely elaborate one level of architectural description in a completely different architectural style. We will see examples of this in the case studies.

4. Case Studies

We now present six examples to illustrate how architectural principles can be used to increase our understanding of software systems. The first example shows how different architectural solutions to the same problem provide different benefits. The second case study summarizes experience in developing a a domain-specific architectural style for a family of industrial products. The third case study examines the familiar compiler architecture in a fresh light. The remaining three case studies present examples of the use of heterogeneous architectures.

4.1. Case Study 1: Key Word in Context

In his paper of 1972, Parnas proposed the following problem [42]:

> The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be ``circularly shifted'' by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Parnas used the problem to contrast different criteria for decomposing a system into modules. He describes two solutions, one based on functional decomposition with shared access to data representations, and a second based on a decomposition that hides design decisions. Since its introduction, the problem has become well-known and is widely used as a teaching device in software engineering. Garlan, Kaiser, and Notkin also use the problem to illustrate modularization schemes based on implicit invocation [7].

While KWIC can be implemented as a relatively small system it is not simply of pedagogical interest. Practical instances of it are widely used by computer scientists. For example, the “permuted” [sic] index for the Unix Man pages is essentially such a system.

From the point of view of software architecture, the problem derives its appeal from the fact that it can be used to illustrate the effect of changes on software design. Parnas shows that different problem decompositions vary greatly in their ability to withstand design changes. Among the changes he considers are:

- Changes in processing algorithm: For example, line shifting can be performed on each line as it is read from the input device, on all the lines after they are read, or on demand when the alphabetization requires a new set of shifted lines.
- Changes in data representation: For example, lines can be stored in various ways. Similarly, circular shifts can be stored explicitly or implicitly (as pairs of index and offset).

Garlan, Kaiser, and Notkin, extend Parnas’ analysis by considering:

- Enhancement to system function: For example, modify the system so that shifted lines to eliminate circular shifts that start with certain noise words (such as “a”, “an”, “and”, etc.). Change the system to be interactive, and allow the user to delete lines from the original (or, alternatively, from circularly shifted) lists.
- Performance: Both space and time.
- Reuse: To what extent can the components serve as reusable entities.

We now outline four architectural designs for the KWIC system. All four are grounded in published solutions (including implementations). The first two are those considered in Parnas’ original article. The third solution is based on the use of an implicit invocation style and represents a variant on the solution examined by Garlan, Kaiser, and Notkin. The fourth is a pipeline solution inspired by the Unix index utility.

After presenting each solution and briefly summarizing its strengths and weakness, we contrast the different architectural decompositions in a table organized along the five design dimensions itemized above.

Solution 1: Main Program/Subroutine with Shared Data

The first solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output. These computational components are coordinated as subroutines by a main program that sequences through them in turn. Data is communicated between the components through shared storage (“core storage”). Communication between the computational components and the shared data is an unconstrained read-write protocol. This is made possible by the fact that the coordinating program guarantees sequential access to the data. (See Figure 6.)

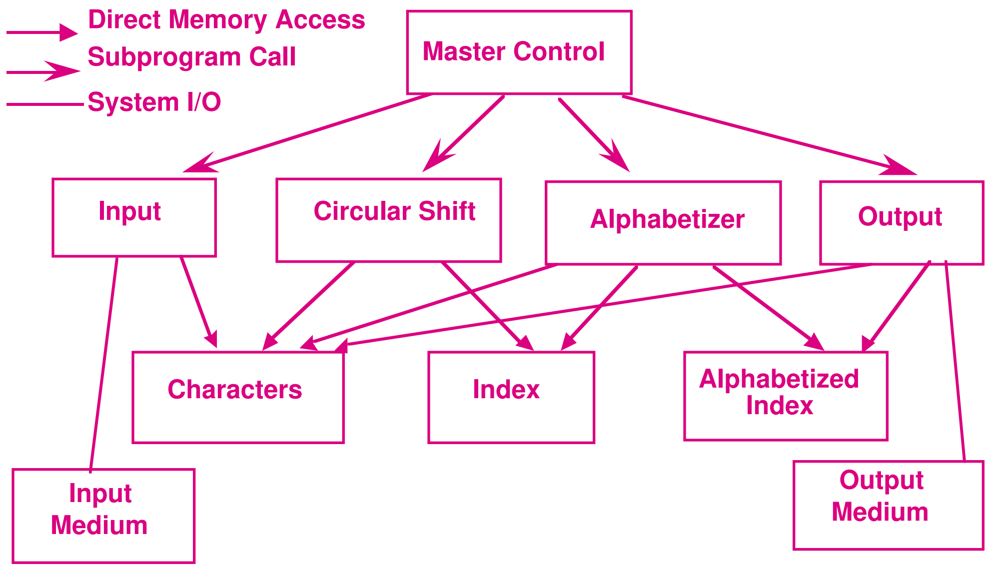

Figure 6: KWIC – Shared Data Solution

Using this solution data can be represented efficiently, since computations can share the same storage. The solution also has a certain intuitive appeal, since distinct computational aspects are isolated in different modules.

However, as Parnas argues, it has a number of serious drawbacks in terms of its ability to handle changes. In particular, a change in data storage format will affect almost all of the modules. Similarly changes in the overall processing algorithm and enhancements to system function are not easily accomodated. Finally, this decom-position is not particularly supportive of reuse.

Solution 2: Abstract Data Types

The second solution decomposes the system into a similar set of five modules. However, in this case data is no longer directly shared by the computational components. Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface. (See Figure 7, which illustrates how each of the components now has a set of procedures that determine the form of access by other components in the system.)

This solution provides the same logical decomposition into processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered. In particular, both algorithms and data representations can be changed in individual modules without affecting others. Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.

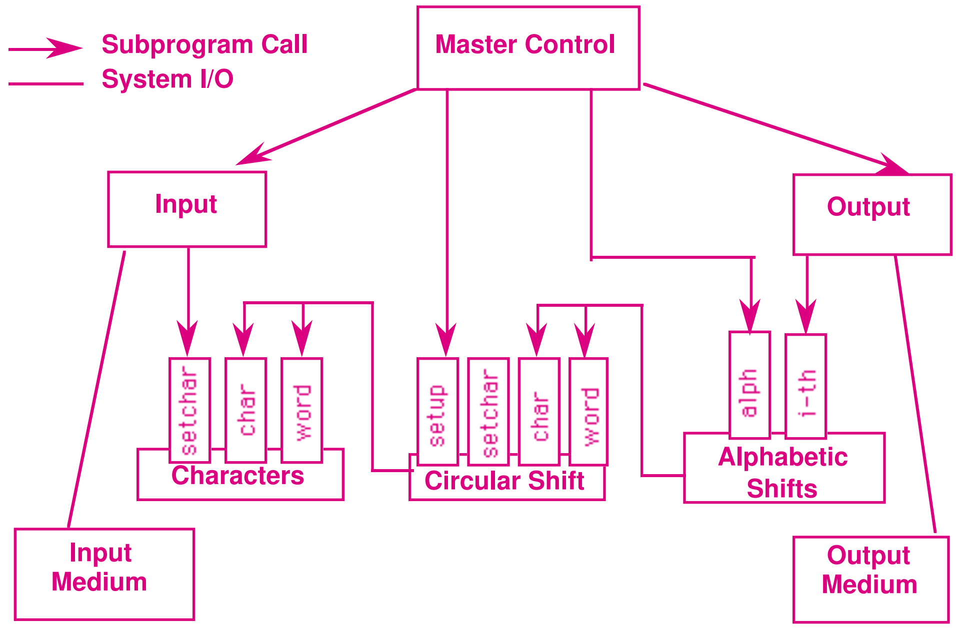

Figure 7: KWIC – Abstract Data Type Solution

On the other hand, as discussed by Garlan, Kaiser, and Notkin, the solution is not particularly well-suited to enhancements. The main problem is that to add new functions to the system, the implementor must either modify the existing modules—compromising their simplicity and integrity—or add new modules that lead to performance penalties. (See [7] for a detailed discussion.)

Solution 3: Implicit Invocation

The third solution uses a form of component integration based on shared data similar to the first solution. However, there are two important differences. First, the interface to the data is more abstract. Rather than exposing the storage formats to the computing modules, data is accessed abstractly (for example, as a list or a set). Second, computations are invoked implicitly as data is modified. Thus interaction is based on an active data model. For example, the act of adding a new line to the line storage causes an event to be sent to the shift module. This allows it to produce circular shifts (in a separate abstract shared data store). This in turn causes the alphabetizer to be implicitly invoked so that it can alphabetize the lines.

This solution easily supports functional enhancements to the system: additional modules can be attached to the system by registering them to be invoked on data-changing events. Because data is accessed abstractly, it also insulates computations from changes in data representation. Reuse is also supported, since the implicitly invoked modules only rely on the existence of certain externally triggered events.

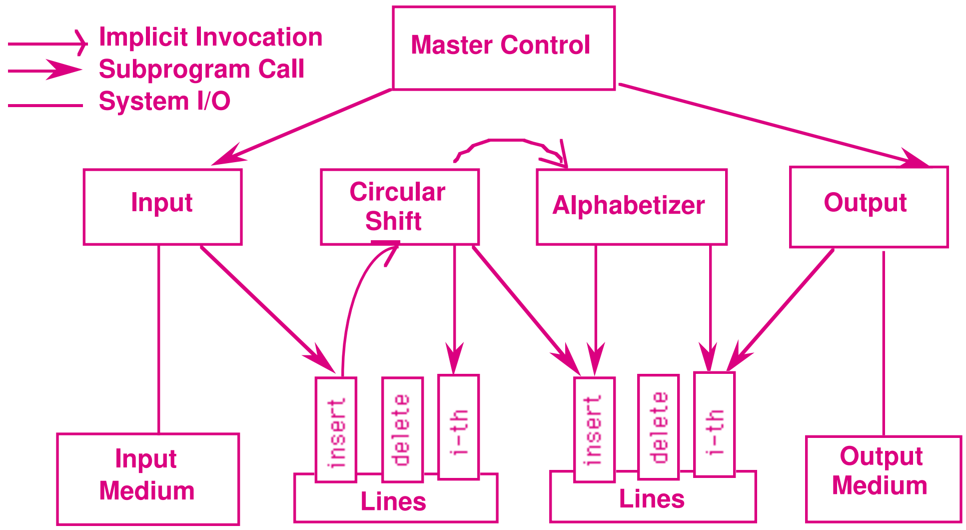

Figure 8: KWIC – Implicit Invocation Solution

However, the solution suffers from the fact that it can be difficult to control the order of processing of the implicitly invoked modules. Further, because invocations are data driven, the most natural implementations of this kind of decomposition tend to use more space than the previously considered decompositions.

Solution 4: Pipes and Filters

The fourth solution uses a pipeline solution. In this case there are four filters: input, shift, alphabetize, and output. Each filter processes the data and sends it to the next filter. Control is distributed: each filter can run whenever it has data on which to compute. Data sharing between filters is strictly limited to that transmitted on pipes. (See Figure 9.)

This solution has several nice properties. First, it maintains the intuitive flow of processing. Second, it supports reuse, since each filter can function in isolation (provided upstream filters produce data in the form it expects). New functions are easily added to the system by inserting filters at the appropriate point in the processing sequence. Third, it supports ease of modification, since filters are logically independent of other filters.

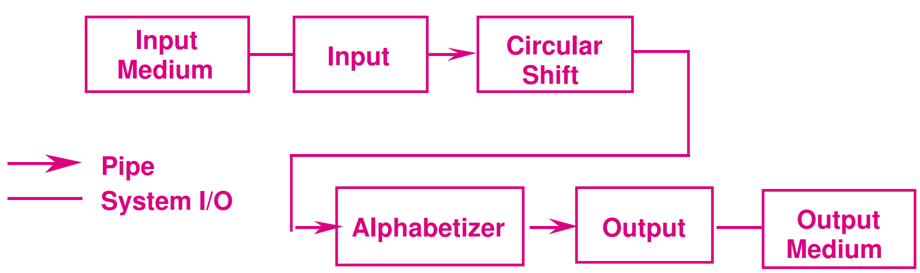

Figure 9: KWIC – Pipe and Filter Solution

On the other hand it has a number of drawbacks. First, it is virtually impossible to modify the design to support an interactive system. For example, in order to delete a line, there would have to be some persistent shared storage, violating a basic tenet of this approach. Second, the solution is inefficient in terms of its use of space, since each filter must copy all of the data to its output ports.

Comparisons

The solutions can be compared by tabulating their ability to address the design considerations itemized earlier. A detailed comparison would have to involve consideration of a number of factors concerning the intended use of the system: for example, is it batch or interactive, update-intensive or query-intensive, etc.

Figure 10 provides an approximation to such an analysis, based on the discussion of architectural styles introduced earlier. As Parnas pointed out, the shared data solution is particularly weak in its support for changes in the overall processing algorithm, data representations, and reuse. On the other hand it can achieve relatively good performance, by virtue of direct sharing of data. Further, it is relatively easy to add a new processing component (also accessing the shared data). The abstract data type solution allows changes to data representation and supports reuse, without necessarily compromising performance. However, the interactions between components in that solution are wired into the modules themselves, so changing the overall processing algorithm or adding new functions may involve a large number of changes to the existing system.

	Shared Memory	ADT	Events	Dataflow
Change in Algorithm	_	_	+	+
Change in Data Repn	_	+	_	_
Change in Function	+	_	+	+
Performance	+	+	_	_
Reuse	_	+	_	+

Figure 10: KWIC – Comparison of Solutions

The implicit invocation solution is particularly good for adding new functionality. However, it suffers from some of the problems of the shared data approach: poor support for change in data representation and reuse. Moreover, it may introduce extra execution overhead. The pipe and filter solution allows new filters to be placed in the stream of text processing. Therefore it supports changes in processing algorithm, changes in function, and reuse. On the other hand, decisions about data representation will be wired into the assumptions about the kind of data that is transmitted along the pipes. Further, depending on the exchange format, there may be additional overhead involved in parsing and unparsing the data onto pipes.

4.2. Case Study 2: Instrumentation Software

Our second case study describes the industrial development of a software architecture at Tektronix, Inc. This work was carried out as a collaborative effort between several Tektronix product divisions and the Computer Research Laboratory over a three year period [6].

The purpose of the project was to develop a reusable system architecture for oscilloscopes. An oscilloscope is an instrumentation system that samples electrical signals and displays pictures (called traces) of them on a screen. Additionally, oscilloscopes perform measurements on the signals, and also display these on the screen. While oscilloscopes were once simple analogue devices involving little software, modern oscilloscopes rely primarily on digital technology and have quite complex software. It is not uncommon for a modern oscilloscope to perform dozens of measurements, supply megabytes of internal storage, interface to a network of workstations and other instruments, and provide sophisticated user interface including a touch panel screen with menus, built-in help facilities, and color displays.

Like many companies that have had to rely increasingly on software to support their products, Tektronix was faced with number of problems. First, there was little reuse across different oscilloscope products. Instead, different oscilloscopes were built by different product divisions, each with their own development conventions, software organization, programming language, and development tools. Moreover, even within a single product division, each new oscilloscope typically required a redesign from scratch to accommodate changes in hardware capability and new requirements on the user interface. This problem was compounded by the fact that both hardware and interface requirements were changing increasingly rapidly. Furthermore, there was a perceived need to address "specialized markets". To do this it would have to be possible to tailor a general-purpose instrument, to a specific set of uses.

Second, there were increasing performance problems because the software was not rapidly configurable within the instrument. This problem arises because an oscilloscope can be configured in many different modes, depending on the user's task. In old oscilloscopes reconfiguration was handled simply by loading different software to handle the new mode. But as the total size of software was increasing, this was leading to delays between a user's request for a new mode and a reconfigured instrument.

The goal of the project was to develop an architectural framework for oscilloscopes that would address these problems. The result of that work was a domain-specific software architecture that formed the basis of the next generation of Tektronix oscilloscopes. Since then the framework has been extended and adapted to accommodate a broader class of system, while at the same time being better adapted to the specific needs of instrumentation software.

In the remainder of this section, we outline the stages in this architectural development.

An object-oriented model

The first attempt at developing a reusable architecture focused on producing an object-oriented model of the software domain. This led to a clarification of the data types used in oscilloscopes: waveforms, signals, measurements, trigger modes, etc. (See Figure 11.)

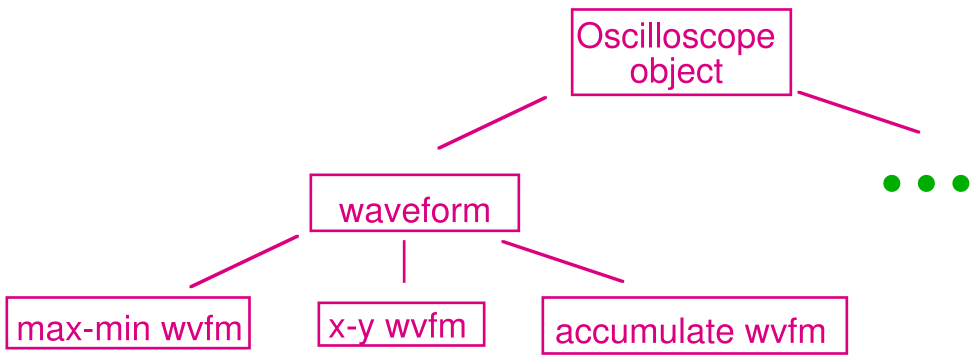

Figure 11: Oscilloscopes – An Object-oriented Model

While this was a useful exercise, it fell far short of producing the hoped-for results. Although many types of data were identified, there was no overall model that explained how the types fit together. This led to confusion about the partitioning of functionality. For example, should measurements be associated with the types of data being measured, or represented externally? Which objects should the user interface talk to?

A layered model

The second phase attempted to correct these problems by providing a layered model of an oscilloscope. (See Figure 11.) In this model the core layer represented the signal manipulation functions that filter signals as they enter the oscilloscope. These functions are typically implemented in hardware. The next layer represented waveform acquisition. Within this layer signals are digitized and stored internally for later processing. The third layer consisted of waveform manipulation, including measurement, waveform addition, Fourier transformation, etc. The fourth layer consisted of display functions. This layer was responsible for mapping digitized waveforms and measurements to visual representations. The outermost layer was the user interface. This layer was responsible for interacting with the user and for deciding which data should be shown on the screen. (See Figure 12.)

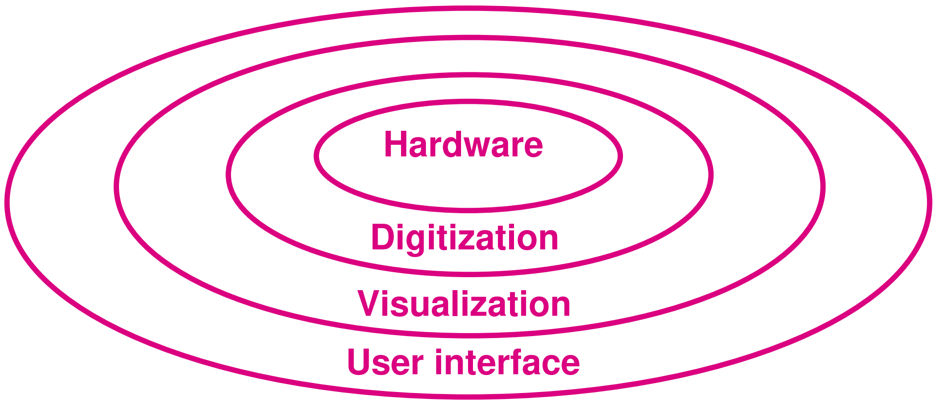

Figure 12: Oscilloscopes – A Layered Model

This layered model was intuitively appealing since it partitioned the functions of an oscilloscope into well-defined groupings. Unfortunately it was the wrong model for the application domain. The main problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction between the various functions. For example, the model suggests that all user interactions with an oscilloscope should be in terms of the visual representations. But in practice real oscilloscope users need to directly affect the functions in all layers, such as setting attenuation in the signal manipulation layer, choosing acquisition mode and parameters in the acquisition layer, or creating derived waveforms in the waveform manipulation layer.

A Pipe and Filter Model

The third attempt yielded a model in which oscilloscope functions were viewed as incremental transformers of data. Signal transformers serve to condition external signals. Acquisition transformers derive digitized waveforms from these signals. Display transformers convert these waveforms into visual data. (See Figure 13.)

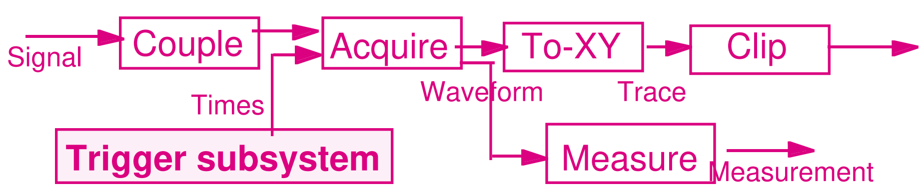

Figure 13: Oscilloscopes – A Pipe and Filter Model

This architectural model was a significant improvement over the layered model in that it did not isolate the functions in separate partitions. For example, nothing in this model would prevent signal data directly feeding into display filters. Further, the model corresponded well to the engineers' view of signal processing as a dataflow problem. The main problem with the model was that it was not clear how the user should interact with it. If the user were simply at one end of the system, then this would represent an even worse decomposition than the layered system.

A Modified Pipe and Filter Model

The fourth solution accounted for user inputs by associating with each filter a control interface that allows an external entity to set parameters of operation for the filter. For example, the acquisition filter might have parameters that determine sample rate and waveform duration. These inputs serve as configuration parameters for the oscilloscope. Formally, the filters can be modelled as "higher-order" functions, for which the configuration parameters determine what data transformation the filter will perform. (See [17] for this interpretation of the architecture.) Figure 14 illustrates this architecture.

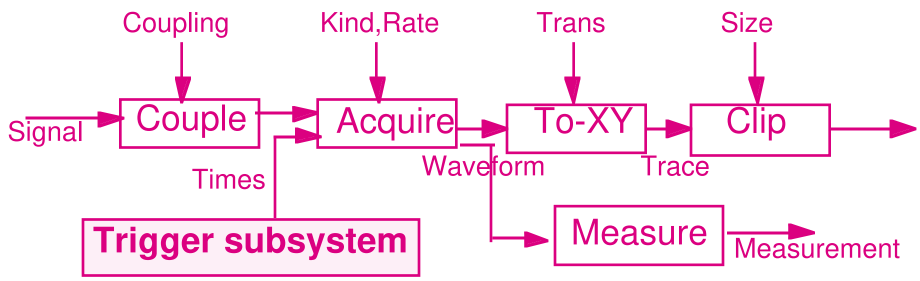

Figure 14: Oscilloscopes – A Modified Pipe and Filter Model

The introduction of a control interface solves a large part of the user interface problem. First, it provides a collection of settings that determine what aspects of the oscilloscope can be modified dynamically by the user. It also explains how changes to oscilloscope function can be accomplished by incremental adjustments to the software. Second it decouples the signal processing functions of the oscilloscope from the actual user interface: the signal processing software makes no assumptions about how the user actually communicates changes to its control parameters. Conversely, the actual user interface can treat the signal processing functions solely in terms of the control parameters. This allowed the designers to change the implementation of the signal processing software and hardware without impacting an interface, provided the control interface remained unchanged.

Further Specialization

The adapted pipe and filter model was a great improvement. But it, too, had some problems. The most significant problem was that the pipe and filter computational model led to poor performance. In particular, waveforms can occupy a large amount of internal storage: it is simply not practical for each filter to copy waveforms every time they process them. Further, different filters may run at radically different speeds: it is unacceptable to slow one filter down because another filter is still processing its data.

To handle these problems the model was further specialized. Instead of having a single kind of pipe, several "colors" of pipes were introduced. Some of these allowed data to be processed without copying. Others permitted data to be ignored by slow filters if they were in the middle of processing other data. These additional pipes increased the stylistic vocabulary and allowed the pipe/filter computations to be tailored more directly to the performance needs of the product.

Summary

This case study illustrates the issues involved in developing an architectural style for a real application domain. It underscores the fact that different architectural styles have different effects on the ability to solve a set of problems. Moreover, it illustrates that architectural designs for industrial software must typically be adapted from pure forms to specialized styles that meet the needs of the specific domain. In this case, the final result depended greatly on the properties of pipe and filter architectures, but found ways to adapt that generic style so that it could also satisfy the performance needs of the product family.

4.3. Case 3: A Fresh View of Compilers

The architecture of a system can change in response to improvements in technology. This can be seen in the way we think about compilers.

In the 1970s, compilation was regarded as a sequential process, and the organization of a compiler was typically drawn as in Figure 15. Text enters at the left end and is transformed in a variety of ways—to lexical token stream, parse tree, intermediate code—before emerging as machine code on the right. We often refer to this compilation model as a pipeline, even though it was (at least originally) closer to a batch sequential architecture in which each transformation ("pass") was completed before the next one started.

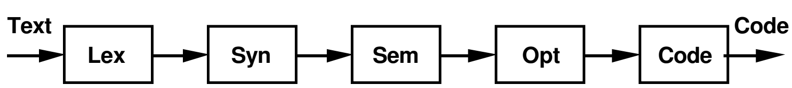

Figure 15: Traditional Compiler Model

In fact, even the batch sequential version of this model was not completely accurate. Most compilers created a separate symbol table during lexical analysis and used or updated it during subsequent passes. It was not part of the data that flowed from one pass to another but rather existed outside all the passes. So the system structure was more properly drawn as in Figure 16.

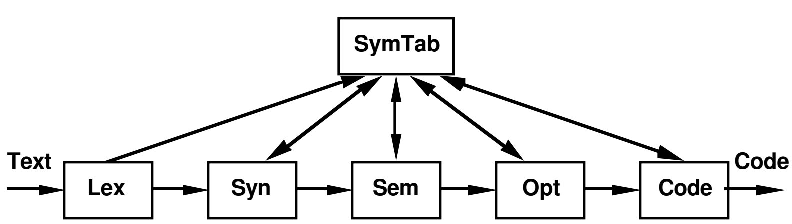

Figure 16: Traditional Compiler Model with Shared Symbol Table

As time passed, compiler technology grew more sophisticated. The algorithms and representations of compilation grew more complex, and increasing attention turned to the intermediate representation of the program during compilation. Improved theoretical understanding, such as attribute grammers, accelerated this trend. The consequence was that by the mid-1980s the intermediate representation (for example, an attributed parse tree), was the center of attention. It was created early during compilation and manipulated during the remainder; the data structure might change in detail, but it remained substantially one growing structure throughout. However, we continued (sometimes to the present) to model the compiler with sequential data flow as in Figure 17.

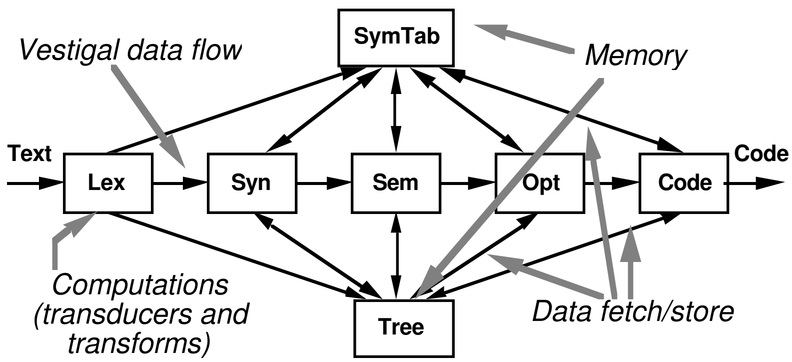

Figure 17: Modern Canonical Compiler

In fact, a more appropriate view of this structure would re-direct attention from the sequence of passes to the central shared representation. When you declare that the tree is the locus of compilation information and the passes define operations on the tree, it becomes natural to re-draw the architecture as in Figure 18. Now the connections between passes denote control flow, which is a more accurate depiction; the rather stronger connections between the passes and the tree/symbol table structure denote data access and manipulation. In this fashion, the architecture has become a repository, and that is indeed a more appropriate way to think about a compiler of this class.

Happily, this new view also accommodates various tools that operate on the internal representation rather than the textual form of a program; these include syntax-directed editors and various analysis tools.

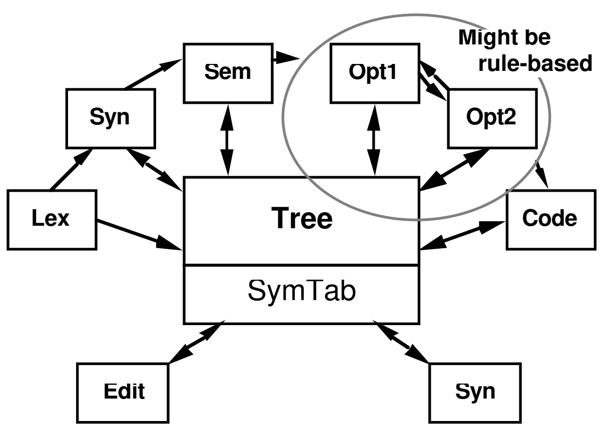

Figure 18: Canonical Compiler, Revisited

Note that this repository resembles a blackboard in some respects and differs in others. Like a blackboard, the information of the computation is located centrally and operated on by independent computations which interact only through the shared data. However, whereas the execution order of the operations in a blackboard is determined by the types of the incoming database modifications, the execution order of the compiler is predetermined.

4.4. Case 4: A Layered Design with Different Styles for the Layers

The PROVOX® system by Fisher Controls offers distributed process control for chemical production processes [43]. Process control capabilities range from simple control loops that control pressure, flow, or levels to complex strategies involving interrelated control loops. Provisions are made for integration with plant management and information systems in support of computer integrated manufacturing. The system architecture integrates process control with plant management and information systems in a 5-level layered hierarchy. Figure 19 shows this hierarchy: the right side is the software view, and the left side is the hardware view.

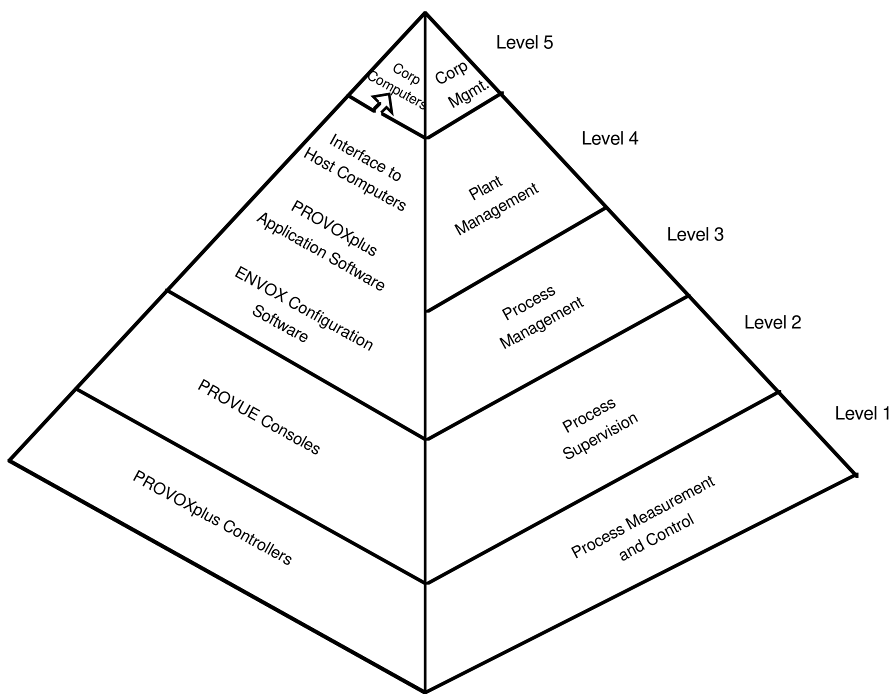

Figure 19: PROVOX – Hierarchical Top Level

Each level corresponds to a different process management function with its own decision-support requirements.

- Level 1: Process measurement and control: direct adjustment of final control elements.
- Level 2: Process supervision: operations console for monitoring and controlling Level 1.

- Level 3: Process management: computer-based plant automation, including management reports, optimization strategies, and guidance to operations console.
- Levels 4 and 5: Plant and corporate management: higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

Different kinds of computation and response times are required at different levels of this hierarchy. Accordingly, different computational models are used. Levels 1 to 3 are object-oriented; Levels 4 and 5 are largely based on conventional data-processing repository models. For present purposes it suffices to examine the object-oriented model of Level 2 and the repositories of Levels 4 and 5.

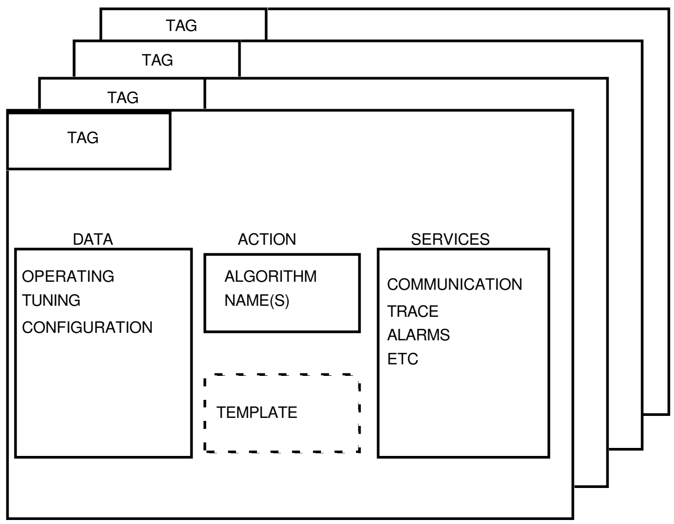

Figure 20: PROVOX – Object-oriented Elaboration

For the control and monitoring functions of Level 2, PROVOX uses a set of points, or loci of process control. Figure 20 shows the canonical form of a point definition; seven specialized forms support the most common kinds of control. Points are, in essence, object-oriented design elements that encapsulate information about control points of the process. The points are individually configured to achieve the desired control strategy. Data associated with a point includes: Operating parameters, including current process value, setpoint (target value), valve output, and mode (automatic or manual). Tuning parameters, such as gain, reset, derivative, and alarm trip-points. Configuration parameters, including tag (name) and I/O channels.

In addition, the point's data can include a template for a control strategy. Like any good object, a point also includes procedural definitions such as control algorithms, communication connections, reporting services, and trace facilities. A collection of points implements the desired process control strategy through the communication services and through the actual dynamics of the process (e.g., if one point increases flow into a tank, the current value of a point that senses tank level will reflect this change). Although the communication through process state deviates from the usual procedural or message-based control of objects, points are conceptually very like objects in their encapsulation of essential state and action information.

Reports from points appear as input transactions to data collection and analysis processes at higher design levels. The organization of the points into control processes can be defined by the designer to match the process control strategy. These can be further aggregated into Plant Process Areas (points related to a set of equipment such as a cooling tower) and thence into Plant Management Areas (segments of a plant that would be controlled by single operators).

PROVOX makes provisions for integration with plant management and business systems at Levels 4 and 5. Selection of those systems is often independent of process control design; PROVOX does not itself provide MIS systems directly but does provide for integrating a conventional host computer with conventional database management. The data collection facilities of Level 3, the reporting facilities of Level 2, and the network that supports distributed implementation suffice to provide process information as transactions to these databases. Such databases are commonly designed as repositories, with transaction processing functions supporting a central data store---quite a different style from the object-oriented design of Level 2.

The use of hierarchical layers at the top level of a system is fairly common. This permits strong separation of different classes of function and clean interfaces between the layers. However, within each layer the interactions among components are often too intricate to permit strict layering.

4.5. Case 5: An Interpreter Using Different Idioms for the Components

Rule-based systems provide a means of codifying the problem-solving know-how of human experts. These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme. Since these rules are not directly executable by available computers, systems for interpreting such rules must be provided. Hayes-Roth surveyed the architecture and operation of rule-based systems [44].

The basic features of a rule-based system, shown in Hayes-Roth's rendering as Figure 21, are essentially the features of a table-driven interpreter, as outlined earlier.

- The *pseudo-code* to be executed, in this case the knowledge base
- The *interpretation engine*, in this case the rule interpreter, the heart of the inference engine

- The *control state of the interpretation engine*, in this case the rule and data element selector
- The *current state of the program* running on the virtual machine, in this case the working memory.

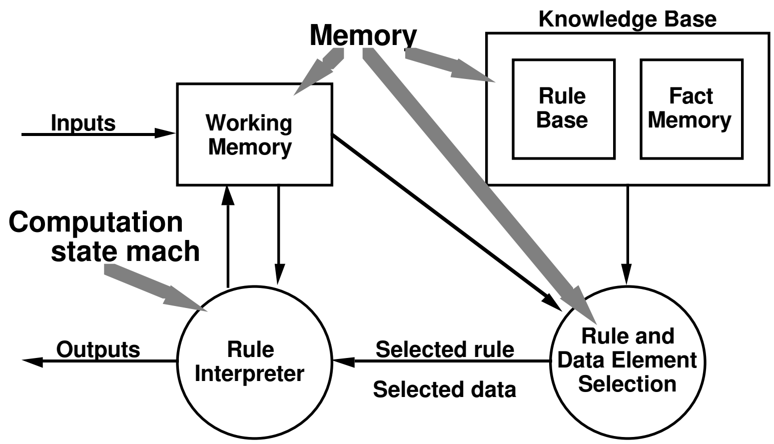

Figure 21: Basic Rule-Based System

Rule-based systems make heavy use of pattern matching and context (currently relevant rules). Adding special mechanisms for these facilities to the design leads to the more complicated view shown in Figure 22. In adding this complexity, the original simple interpreter vanishes in a sea of new interactions and data flows. Although the interfaces among the original modules remain, they are not distinguished from the newly-added interfaces.

However, the interpreter model can be rediscovered by identifying the components of Figure 22 with their design antecedents in Figure 21. This is done in Figure 23. Viewed in this way, the elaboration of the design becomes much easier to explain and understand. For example, we see that:

- The knowledge base remains a relatively simple memory structure, merely gaining substructure to distinguish active from inactive contents.
- The rule interpreter is expanded with the interpreter idiom (that is, the interpretation engine of the rule-based system is itself implemented as a table-driven interpreter), with control procedures playing the role of the pseudo-code to be executed and the execution stack the role of the current program state.
- "Rule and data element selection" is implemented primarily as a pipeline that progressively transforms active rules and facts to prioritized activations; in this pipeline the third filter ("nominators") also uses a fixed database of metarules.
- Working memory is not further elaborated.

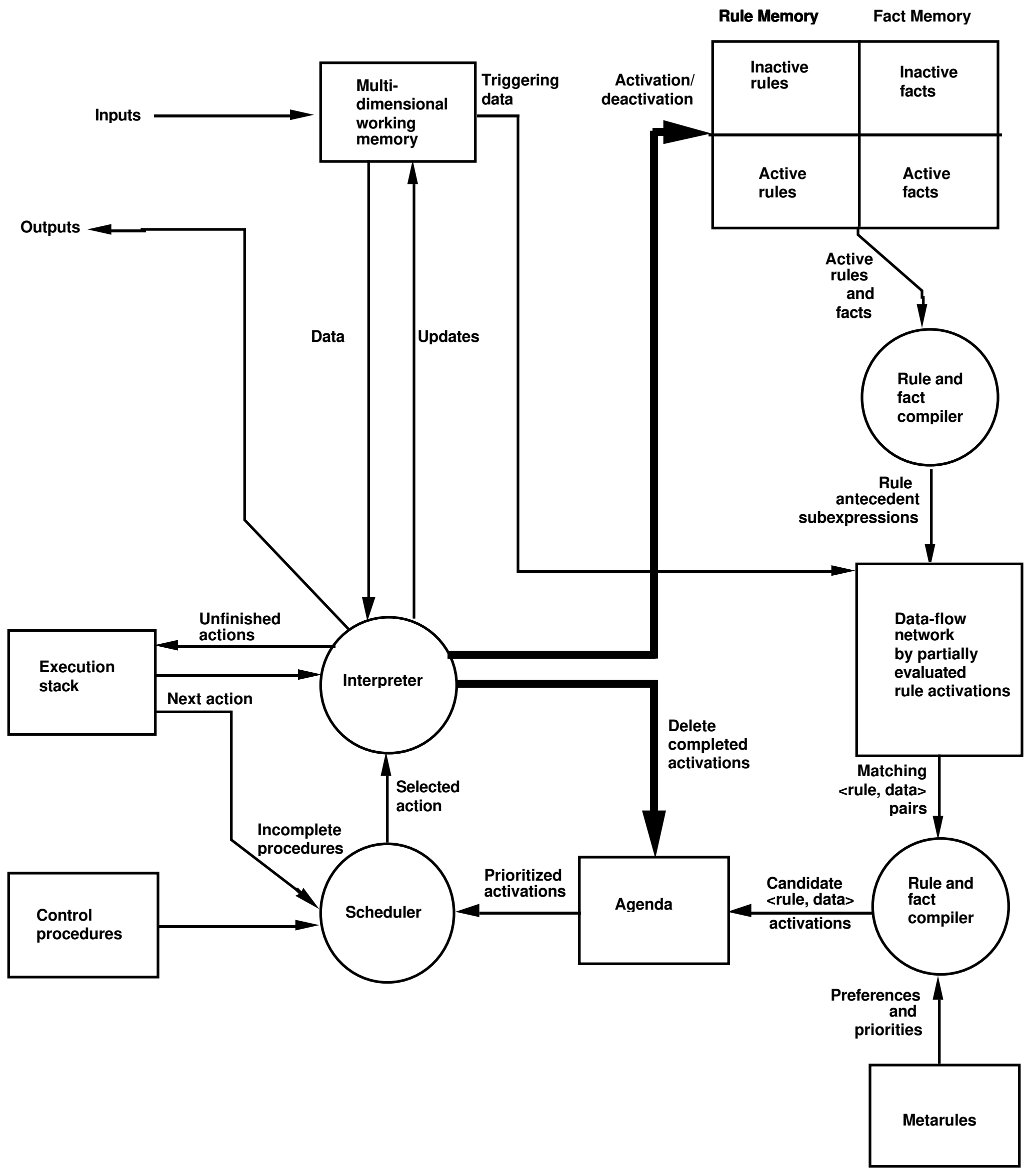

Figure 23: Sophisticated Rule-Based System

The interfaces among the rediscovered components are unchanged from the simple model except for the two bold lines over which the interpreter controls activations.

This example illustrates two points. First, in a sophisticated rule-based system the elements of the simple rule-based system are elaborated in response to execution characteristics of the particular class of languages being interpreted. If the design is presented in this way, the original concept is retained to guide understanding and later maintenance. Second, as the design is elaborated, different components of the simple model can be elaborated with different idioms.

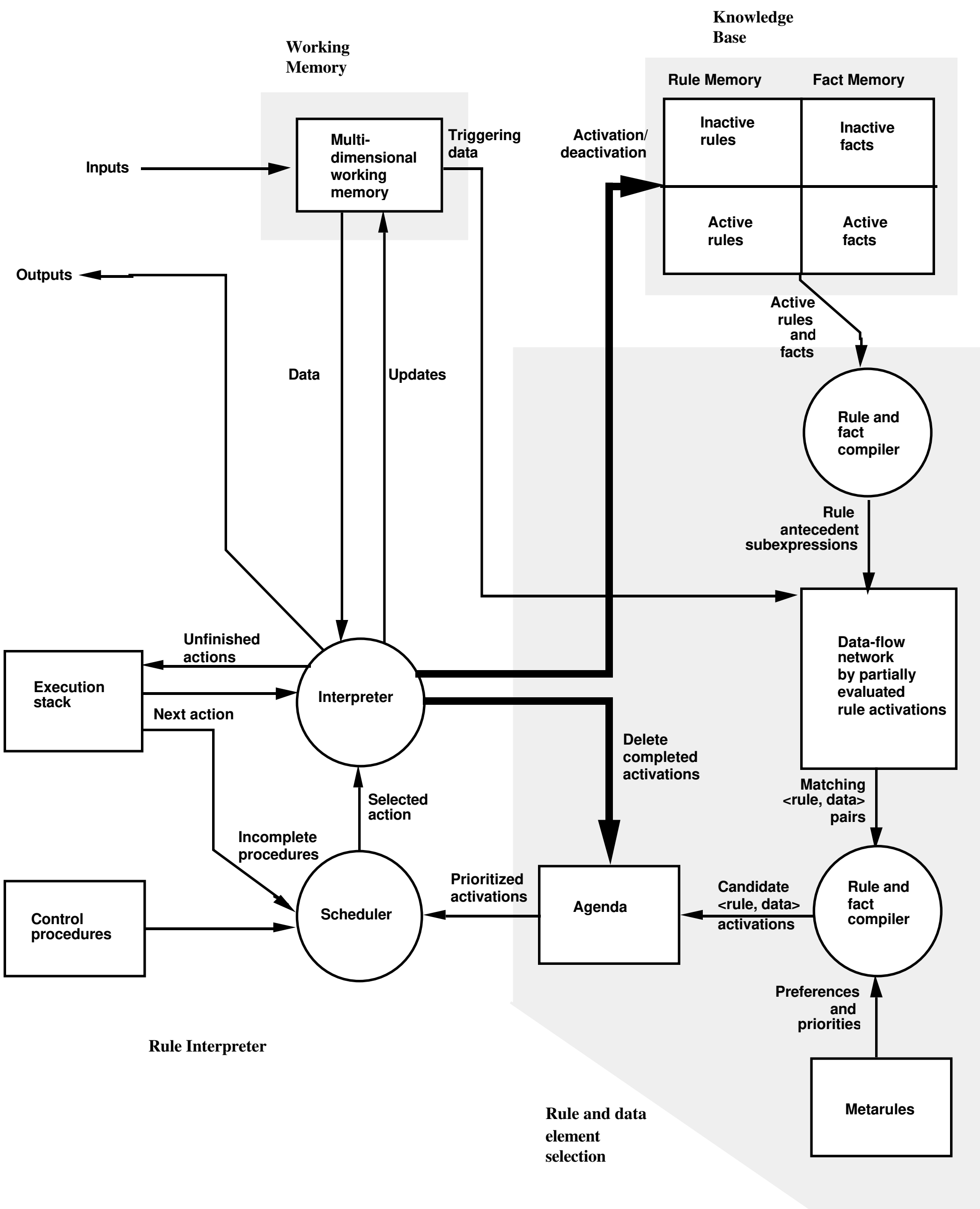

Figure 23: Simplified Sophisticated Rule-Based System

Note that the rule-based model is itself a design structure: it calls for a set of rules whose control relations are determined during execution by the state of the computation. A rule-based system provides a virtual machine—a rule executor—to support this model.

4.6. *Case 6: A Blackboard Globally Recast as Interpreter*

The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. In this model, the solution space is organized into several application-dependent hierarchies and the domain knowledge is partitioned into independent modules of knowledge that operate on knowledge within and between levels [34]. Figure 4 showed the basic architecture of a blackboard system and outlined its three major parts: knowledge sources, the blackboard data structure, and control.

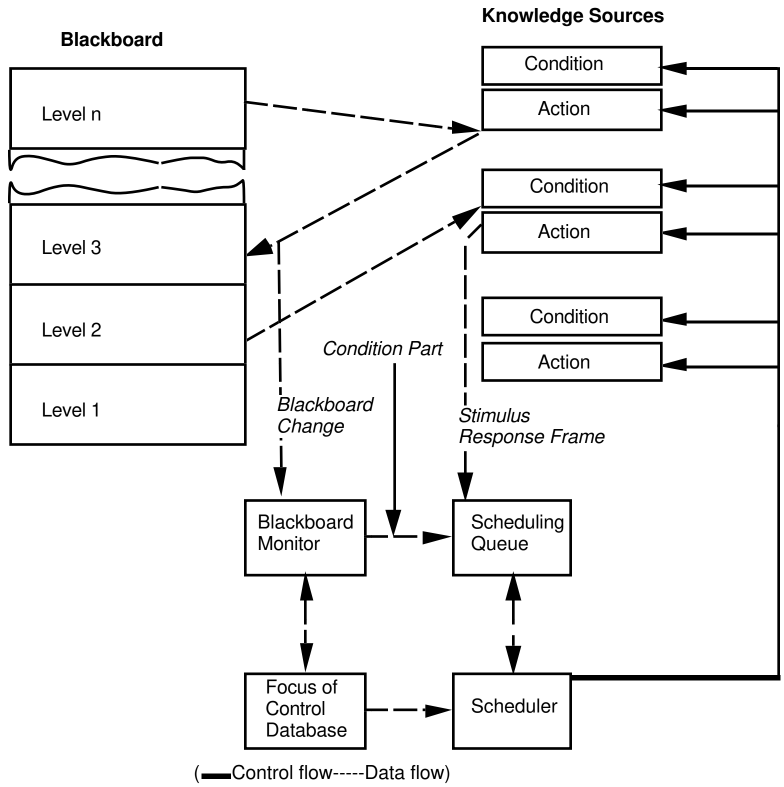

Figure 24: Hearsay-II

The first major blackboard system was the HEARSAY-II speech recognition system. Nii's schematic of the HEARSAY-II architecture appears as Figure 24. The blackboard structure is a six- to eight-level hierarchy in which each level abstracts information on its adjacent lower level and blackboard elements represent hypotheses about the interpretation of an utterance. Knowledge sources correspond to such tasks as segmenting the raw signal, identifying phonemes, generating word candidates, hypothesizing syntactic segments, and proposing semantic interpretations. Each knowledge source is organized as a condition part that specifies when it is applicable and an action part that processes relevant blackboard elements and generates new ones. The control component is realized as a blackboard monitor and a scheduler; the scheduler monitors the blackboard and calculates priorities for applying the knowledge sources to various elements on the blackboard.

HEARSAY-II was implemented between 1971 and 1976 on DEC PDP-10s; these machines were not directly capable of condition-triggered control, so it should not be surprising to find that an implementation provides the mechanisms of a virtual machine that realizes the implicit invocation semantics required by the blackboard model.

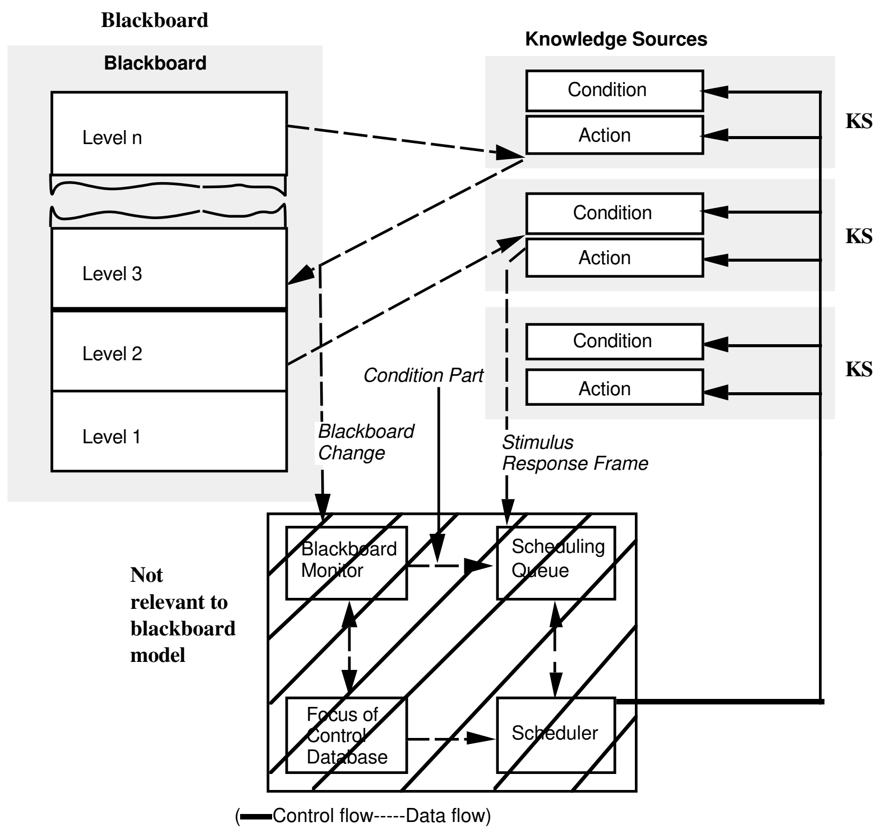

Figure 25: Blackboard View of Hearsay-II

Figure 24 not only elaborates the individual components of Figure 4; it also adds components for the previously-implicit control component. In the process, the figure becomes rather complex. This complexity arises because it is now illustrating two concepts: the blackboard model and realization of that model by a virtual machine. The blackboard model can be recovered as in Figure 25 by suppressing the control mechanism and regrouping the conditions and actions into knowledge sources.

The virtual machine can be seen to be realized by an interpreter using the assignment of function in Figure 26. Here the blackboard cleanly corresponds to the current state of the recognition task. The collection of knowledge sources roughly supplies the pseudocode of the interpreter; however, the actions also contribute to the interpretation engine. The interpretation engine includes several components that appear explicitly in Figure 24: the blackboard monitor, the focus of control database, and the scheduler, but also the actions of the knowledge sources. The scheduling queue corresponds roughly to the control state. To the extent that execution of conditions determines priorities, the conditions contribute to rule selection as well as forming pseudocode.

Here we see a system initially designed with one model (blackboard, a special form of repository), then realized through a different model (interpreter). The realization is not a component-by-component expansion as in the previous two examples; the view as an interpreter is a different aggregation of components from the view as blackboard.

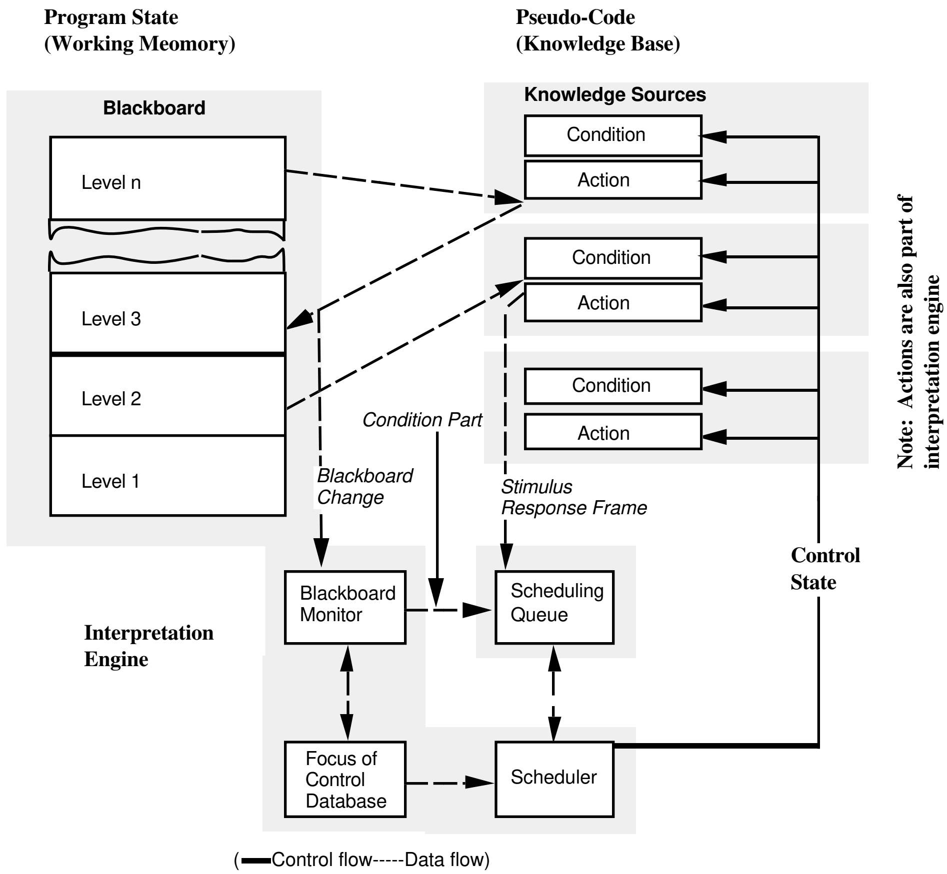

Figure 26: Interpreter View of Hearsay-II

5. Past, Present, and Future

We have outlined a number of architectural styles and shown how they can be applied and adapted to specific software systems. We hope that this has convinced the reader that analysis and design of systems in terms of software architecture is both viable and worth doing. Further we hope to have made it clear that an understanding of the emerging vocabulary of architectural styles is a significant—if not necessary—intellectual tool for the software engineer.

There is, of course, much more to software architecture than we have had space to cover. In particular, we have said very little about existing results in the areas of analysis, formal specification, domain-specific architectures, module interconnection languages, and special-architecture tools.

This is not to say that more work isn't needed. Indeed, we can expect to see significant advances in a number of areas including:

- Better taxonomies of architectures and architectural styles.

- Formal models for characterizing and analyzing architectures.
- Better understanding of the primitive semantic entities from which these styles are composed.
- Notations for describing architectural designs.
- Tools and environments for developing architectural designs.
- Techniques for extracting architectural information from existing code.
- Better understanding of the role of architectures in the life-cycle process.

These are all areas of active research both in industry and academia. Given the increasing interest in this emerging field, we can expect that our understanding of the principles and practice of software architecture will improve considerably over time. However, as we have illustrated, even with the basic concepts that we now have in hand, design at the level of software architecture can provide direct and substantial benefit to the practice of software engineering.

Acknowledgements

We gratefully acknowledge our many colleagues who have contributed to the ideas presented in this paper. In particular, we would like to thank Chris Okasaki, Curtis Scott, and Roy Swonger for their help in developing the course from which much of this material was drawn. We thank David Notkin, Kevin Sullivan, and Gail Kaiser for their contribution towards understanding event-based systems. Rob Allen helped develop a rigorous understanding of the pipe and filter style. We would like to acknowledge the oscilloscope development team at Tektronix, and especially Norm Delisle, for their part in demonstrating the value of domain-specific architectural styles in an industrial context. Finally, we would like to thank Barry Boehm, Larry Druffel, and Dilip Soni for their constructive comments on early drafts of the paper.

Bibliography

[1] D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger, "Experience with a course on architectures for software systems," in *Proceedings of the Sixth SEI Conference on Software Engineering Education,* Springer-Verlag, LNCS 376, October 1992.

[2] M. Shaw, "Toward higher-level abstractions for software systems," in *Data & Knowledge Engineering,* vol. 5, pp. 119-128, North Holland: Elsevier Science Publishers B.V., 1990.

[3] M. Shaw, "Heterogeneous design idioms for software architecture," in *Proceedings of the Sixth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes,* (Como, Italy), pp. 158-165, October 25-26, 1991.

[4] M. Shaw, "Software architectures for shared information systems," in *Mind Matters: Contributions to Cognitive and Computer Science in Honor of Allen Newell,* Erlbaum, 1993.

[5] R. Allen and D. Garlan, "A formal approach to software architectures," in *Proceedings of IFIP'92* (J. van Leeuwen, ed.), Elsevier Science Publishers B.V., September 1992.

[6] D. Garlan and D. Notkin, "Formalizing design spaces: Implicit invocation mechanisms," in *VDM'91: Formal Software Development Methods.* (Noordwijkerhout, The Netherlands), pp. 31-44, Springer-Verlag, LNCS 551, October 1991.

[7] D. Garlan, G. E. Kaiser, and D. Notkin, "Using tool abstraction to compose systems," *IEEE Computer,* vol. 25, June 1992.

[8] A. Z. Spector *et al.*, "Camelot: A distributed transaction facility for Mach and the Internet - an interim report," Tech. Rep. CMU-CS-87-129, Carnegie Mellon University, June 1987.

[9] M. Fridrich and W. Older, "Helix: The architecture of the XMS distributed file system," *IEEE Software,* vol. 2, pp. 21-29, May 1985.

[10] M. A. Linton, "Distributed management of a software database," *IEEE Software*, vol. 4, pp. 70-76, November 1987.

[11] V. Seshadri et al., "Sematic analysis in a concurrent compiler," in *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* ACM SIGPLAN Notices, 1988.

[12] M. C. Paulk, "The ARC Network: A case study," *IEEE Software,* vol. 2, pp. 61-69, May 1985.

[13] M. Chen and R. J. Norman, "A framework for integrated case," IEEE Software, vol. 9, pp. 18-22, March 1992.

[14] NIST/ECMA reference model for frameworks of software engineering environments." NIST Special Publication 500-201, December 1991.

[15] R. W. Scheifler and J. Gettys, "The X window system," *AACM Transactions on Graphics,* vol. 5, pp. 79-109, April 1986.

[16] M. J. Bach, *The Design of the UNIX Operating System,* ch. 5.12, pp. 111-119. Software Series, Prentice-Hall, 1986.

[17] N. Delisle and D. Garlan, "Applying formal specification to industrial problems: A specification of an oscilloscope," *IEEE Software,* September 1990.

[18] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing,* 1974.

[19] M. R. Barbacci, C. B. Weinstock, and J. M. Wing, "Programming at the processor-memory-switch level," in *Proceedings of the 10th International Conference on Software Engineering,* (Singapore), pp. 19-28, IEEE Computer Society Press, April 1988.

[20] G. E. Kaiser and D. Garlan, "Synthesizing programming environments from reusable features," in Software Reusability (T. J. Biggerstaff and A. J. Perlis, eds.), vol. 2, ACM Press, 1989.

[21] W. Harrison, "RPDE: A framework for integrating tool fragments," IEEE Software, vol. 4, November 1987.

[22] C. Hewitt, "Planner: A language for proving theorems in robots," in *Proceedings of the First International Joint Conference in Artificial Intelligence,* 1969.

[23] S. P. Reiss, "Connecting tools using message passing in the field program development environment," *IEEE Software,* July 1990.

[24] C. Gerety, "HP Softbench: A new generation of software development tools," Tech. Rep. SESD-89-25, Hewlett-Packard Software Engineering Systems Division, Fort Collins, Colorado, November 1989.

[25] R. M. Balzer, "Living with the next generation operating system," in *Proceedings of the 4th World Computer Conference,* September 1986.

[26] G. Krasner and S. Pope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80," Journal of Object Oriented Programming, vol. 1, pp. 26-49, August/September 1988.

[27] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A programming-language approach to interactive display interfaces," *Proceedings of SIGPLAN '83: Symposium on Programming Language Issues in Software Systems, ACM SIGPLAN Notices,* vol. 18, pp. 100-111, June 1983.

[28] A. N. Habermann and D. S. Notkin, "Gandalf: Software development environments," *IEEE Transactions on Software Engineering,* vol. SE-12, pp. 1117-1127, December 1986.

[29] A. N. Habermann, D. Garlan, and D. Notkin, "Generation of integrated task-specific software environments," in *CMU Computer Science: A 25th Commemorative* (R. F. Rashid, ed.), Anthology Series, pp. 69-98, ACM Press, 1991.

[30] K. Sullivan and D. Notkin, "Reconciling environment integration and component independence," in *Proceedings of ACM SIGSOFT90: Fourth Symposium on Software Development Environments*, pp. 22-23, December 1990.

[31] G. R. McClain, ed., *Open Systems Interconnection Handbook.* New York, NY: Intertext Publications McGraw-Hill Book Company, 1991.

[32] D. Batory and S. O'Malley, "The design and implementation of hierarchical software systems using reusable components," Tech. Rep. TR-91-22, Department of Computer Science, University of Texas, Austin, June 1991.

[33] H. C. Lauer and E. H. Satterthwaite, "Impact of MESA on system design," in *Proceedings of the Third International Conference on Software Engineering,* (Atlanta, GA), pp. 174-175, IEEE Computer Society Press, May 1979.

[34] H. P. Nii, "Blackboard systems Parts 1 & 2," AI Magazine, vol. 7 nos 3 (pp. 38-53) and 4 (pp. 62-69), 1986.

[35] V. Ambriola, P. Ciancarini, and C. Montangero, "Software process enactment in oikos," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments,* SIGSOFT Software Engineering Notes, (Irvine, CA), pp. 183-192, December 1990.

[36] D. R. Barstow, H. E. Shrobe, and E. Sandewall, eds., *Interactive Programming Environments.* McGraw-Hill Book Co., 1984.

[37] G. R. Andrews, "Paradigms for process interaction in distributed programs," *ACM Computing Surveys,* vol. 23, pp. 49-90, March 1991.

[38] A. Berson, *Client/Server Architecture.* McGraw-Hill, 1992.

[39] E. Mettala and M. H. Graham, eds., *The Domain-Specific Software Architecture Program.* No. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[40] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming,* vol. 8, pp. 231-274, 1987.

[41] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems Design.* Prentice Hall, second ed., 1990.

[42] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM,* vol. 15, pp. 1053-1058, December 1972.

[43] "PROVOX plus Instrumentation System: System overview," 1989.

[44] F. Hayes-Roth, 'Rule-based systems," Communications of the ACM, vol. 28, pp. 921-932, September 1985.