1. Introduction

Software architecture is concerned with system structure—organization of the software, assignment of responsibilities to components, and assurance that the components' interactions satisfy the system requirements [GS93, PW92]. Software developers recognize a number of distinct architectural styles. Many of these styles are defined informally and idiosyncratically. Our purpose here is to clarify the distinctions among styles as a first step in helping designers choose among the styles.

By *architectural style* we mean a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done. *Components,* including encapsulated subsystems, may be distinguished by the nature of their computation (e.g., whether they retain state from one invocation to another, and if so, whether that state is available to other components). Component types may also be distinguished by their packaging—the ways they interact with other components. Packaging is usually implicit, which tends to hide important properties of the components. To clarify the abstractions we isolate the definitions of these interaction protocols in *connectors* (e.g., processes interact via message-passing protocols; unix filters interact via data flow through pipes). It is largely the interaction among components, mediated by connectors, that gives different styles their distinctive characteristics.

The style of a specific system is usually established by appeal to common knowledge or intuition. Architectures are usually expressed in box-and-line diagrams and informal prose, so the styles provide drawing conventions, vocabulary, and informal constraints (e.g., limiting topology or numbers of components of some type). Recently there has been some effort to identify and define styles more precisely and systematically [G+94, SG96, Sh96]. A few styles have been formalized or extensively analyzed [A+95, AG94, An91, Ni86]. Space does not permit us to offer primary definitions of specific styles here.

In this paper we begin to organize and classify some of the styles that appear in software descriptions. By doing so we aim to

- Establish a *uniform descriptive standard* for architectural styles—make the vocabulary used to describe styles more precise and shareable among software architects.
- Provide a systematic *organization to support retrieval* of information about styles.
- *Discriminate* among different styles—bring out significant differences that affect the suitability of a style for various tasks; show which styles are refinements of others.
- Set the stage for organizing advice on *selecting a style* for a given problem.

This classification is an intermediate step toward supporting architectural design decisions. Minimally, it will help the designer focus on important design issues by providing a checklist of topics to consider, thereby setting expectations for elements to include in the design. Eventually the classification should provide guidance for recognizing which styles are important candidates for shaping the solution.

This work shares motivation with recent work on problem frames and on patterns. Jackson's work on problem frames [Ja96] does for classes of problems what we are doing for classes of solutions. Jackson distinguishes classes of problems on the basis of the kinds of information provided and kinds of results expected. In the space of solutions, the patterns community concentrates on documenting proven solutions, including architectural styles, in the context of the specific kinds of problem for which each solution is useful [GoF95, CS95, Sh96]. Both

Jackson and the patterns community begin with observation of things that work and work toward principled models and guidance. The present work is precisely in that spirit.

Much of computer science includes phenomena about which we do not yet have well-established scientific truths but do have interesting observations and generalizations. Brooks has proposed recognizing three kinds of results: *findings* (well-established scientific truths), *observations* (reports on actual phenomena), and *rules-of-thumb* (generalizations, signed by an author but perhaps not fully supported by data). His criteria for judging quality are truthfulness and rigor for *findings*, interestingness for *observations*, usefulness for *rules-of-thumb*, and freshness for all three [Br88]. This paper presents observations (on discriminating among styles) and some rules-of-thumb (guiding their use in design). We have based our classification on an extensive set of system descriptions. The work is empirical and observational—we try to account for the descriptions that designers actually write, as opposed to inventing styles that may be easy to classify but are of no apparent practical import.

Section 2 presents the classification of architectural styles, using feature categories explained in Section 3. Section 4 shows that the classification is feasible extensible by using it to elaborate variants of two major styles that have appeared in the literature. Section 5 discusses the role of style in design, including style support in architecture description languages, and provides some rules of thumb for choosing a style on the basis of the problem at hand.

2. Styles for Software Architectures

In this section we classify a set of architectural styles that have been described previously (usually informally) in published literature. Software designers use an extensive descriptive vocabulary to explain their system organizations. They use the vocabulary informally, almost casually. Often a single label or graphical element is used with different meanings. Through a rigorous classification we attempt to capture the common meanings.

Table 1 shows the resulting classification. It is not complete, but it spans much of the diversity found in practice. Each row describes a style. Columns correspond to the feature categories as described in Section 3. Indented rows describe specializations of the styles in the primary rows they follow. Because this is a multidimensional classification, it is possible for a style to be a variant of more than one broader style; we handle this with cross-references.

We describe the styles in their pure forms, although they seldom occur that way. Real systems hybridize and amalgamate the pure styles, with the architect choosing useful aspects from several in order to accomplish the task at hand. Our classification does not impede this heterogeneity, but rather enhances the selection and blending process by making stylistic properties explicit. Understanding the pure forms is helpful in understanding or explaining the hybrids, and perhaps also in recognizing and eliminating unnecessary heterogeneity. Indeed, the classification activity can point out common styles that are hybrids of other styles, such as lightweight processes.

After looking through the table many readers will say, “But that’s not what *I* mean by style X!”. Indeed, it may not be. But it is, as far as we can tell, what *someone else* means. This is an indication that different readers use style names in different ways. A primary objective of this classification is to expose these differences and enable constructive discussion.

3. Classification Strategy

A system designer's primary impression of an architecture often keys on the character of the interactions among components. Our classification strategy reflects this. The major axes of classification are the control and data interactions among components. We make finer discriminations within these dimensions.

Our analysis of common architectural styles suggests that they are discriminated by the following categories of features:

- which kinds of components and connectors are used in the style
- how control is shared, allocated, and transferred among the components
- how data is communicated through the system
- how data and control interact
- what type of reasoning is compatible with the style

These categories form the basis of a descriptive classification that shows essential similarities and differences among styles. Features that distinguish styles also help us understand why a particular style is an appropriate solution for one type of problem and not for another.

This classification is based on coarse-grained descriptions of properties, a technique often used for qualitative descriptions. It allows significant, yet comprehensible, distinctions to be made without risk of combinatorial explosion. Finer distinctions can be made, when appropriate, in refinements of the primary analysis; this is done for two examples in Section 4. It is useful to think of each of these feature categories as defining one dimension of a multidimensional space [La90]. A specific style then corresponds to a point in the space and a family of styles corresponds to a subspace.

In a multidimensional space, it is possible for two subspaces to intersect. For example, lightweight processes, which are usually thought of as a variant of communicating processes, share a name space. Since this is added capability, they are not a specialization (i.e., subspace) of processes. The shared name space enables data sharing, which positions lightweight processes as a variant of shared data as well. Design with lightweight processes must therefore recognize the problems and advantages of both parents. The overlap is reflected in Table 1.

An alternative way to distinguish styles would be through a hierarchical taxonomy [PN86]. Such taxonomies are structured as decision trees with the taxonomized population at the leaves and discriminating questions on the interior nodes. Such taxonomies may be more expressive than tabular classifications because they can use different discrimination criteria at different points. However, they show similarities less well than design spaces do, because they can describe only the families that correspond to subtrees, not families along some other dimension (or several dimensions). Section 4 shows how we can use the additional descriptive power of the design space to explore regions of it in detail.

The following sections detail each feature category.

3.1 Constituent Parts: Components and connectors

Components and connectors are the primary building blocks of architectures. A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters. A connector is a mechanism that mediates communication,

coordination, or cooperation among components. Implementations of connectors are usually distributed over many system components; often they do not correspond to discrete elements of the running system. Examples include shared representations, remote procedure calls, message-passing protocols, data streams, and transaction streams.

We focus on the *abstractions* used by designers in defining their architectures. In practice, most of these elements are ultimately *implemented* in terms of processes (as defined by the operating system) and procedure calls (as defined by the programming language). More abstract connectors include format conversions that allow two otherwise-incompatible components to share data and connectors augmented by performance monitoring, authentication, or audit-trail capabilities.

The allowable kinds of components and connectors are primary discriminators among styles. Selecting the types of constituent parts does not, however, uniquely identify the style. Control disciplines, data organizations, and the interaction of control and data all affect style distinctions. So do finer distinctions within types of components and connectors, some of which appear in Table 1. For example, both *program* and *transducer* refine *process*; *procedure calls* may be *local* or *remote*, and their binding may be *dynamic* or *static*; *batch data*, *data stream*, and *continuous refresh* are all forms of *data flow*.

A taxonomic treatment of architectural components and connectors, filling out the conceptual framework begun here, appears elsewhere [K+96].

Components and connectors also provide a clustering criterion for the styles in Table 1. Members of a cluster share similar types of components and/or connectors. These clusters are not the only ones possible; the styles could be grouped differently by choosing other features as their organizing basis. Indeed, different groupings may be desirable, especially if they correspond to the language of a problem description. However, the component/connector-based clusters enjoy a certain intuitive clarity, reflect how many practitioners tend to describe styles, and mirror previous *a priori* classification efforts [GS93].

3.2 Control issues

Control issues describe how control passes among components and how the components work together temporally. Control issues include:

- **Topology**: What geometric form does the control flow for the system take? A pipeline often has a *linear* (non-branching) or at least an *acyclic* control topology; a main-program-and-subroutines style features a *hierarchical* (tree-shaped) topology; some server systems have *star* (hub-and-spoke) topologies; a style consisting of communicating sequential processes may have an *arbitrary* topology. Some architectures specify quite specific topologies. Within each general topology it may be useful to stipulate the direction in which control flows. The topology may be static or dynamic; this is determined by the binding time of the partner as described below.
- **Synchronicity**: How dependent are the components' actions upon each others' control states? In a *lockstep* system, the state of any component implies the state of all others; for instance, a batch sequential system's components are in lockstep with each other, since one doesn't begin execution until its predecessor finishes. SIMD (same instruction, multiple data) algorithms for massively parallel machines also work in lockstep. In *synchronous* systems, components synchronize regularly and often, but other state relationships are unpredictable. *Asynchronous* components are largely unpredictable

in their interaction or synchronize once in a while, while *opportunistic* components such as autonomous agents work completely independently from each other in parallel.

Lockstep systems can be *sequential* or *parallel*, depending on how many threads of control run through them. Other forms of synchronicity imply parallelism.

- **Binding time**: When is the identity of a partner in a transfer-of-control operation established? Some control transfers are pre-determined at program-*write* (i.e., source code) time, *compile* time, or *invocation* time (i.e., when the operating system initializes the process). Others are bound dynamically while the system is *run*ning.

3.3 Data issues

Data issues describe how data moves around a system. Data issues include:

- **Topology**: Data topology describes the geometric shape of the system's data flow graph. The alternatives are as for control topology
- **Continuity**: How continuous is the flow of data throughout the system? A *continuous*-flow system has fresh data available at all times; a *sporadic*-flow system has new data generated at discrete times. Data transfer may also be *high-volume* (in data-intensive systems) or *low-volume* (in compute-intensive systems).
- **Mode**: Data mode describes how data is made available throughout the system. In an object style, it is *passed* from component to component, whereas in any of the shared data systems it is *shared* by making it available in a place accessible to all the sharers. If the components tend to modify it and re-insert it into the public store, this is a *copy-out-copy-in* mode. In some styles data is *broadcast* or *multicast* to specific recipients.
- **Binding time**: When is the identity of a partner in a transfer-of-control operation established? This is the data analogy of the same control issue,

3.4 Control/data interaction issues

Interaction issues describe the relationship between certain control and data issues.

- **Shape**: Are the control flow and data flow topologies substantially isomorphic to each other?
- **Directionality**: If the shapes are substantially the same, does control flow in the *same* direction as data or the *opposite* direction? In a data-flow system such as pipe-and-filter, control and data pass together from component to component. However, in a client-server style, control tends to flow into the servers while data flows into the clients.

3.5 Type of reasoning

Different classes of architectures lend themselves to different types of analysis. A system of components operating asynchronously in parallel yields to vastly different reasoning approaches (e.g., nondeterministic state machine theory) than a system that executes as a fixed sequence of atomic steps (e.g., function composition). Many analysis techniques compose their results from analysis of substructures, but this depends on the ability to combine sub-analyses. The fit of an analysis technique to an architecture is enhanced if the software organization matches the analysis organization—that is, if software substructure and analy-

sis substructure are compatible.

Thus, different architectural styles are good matches for different analysis techniques. Your choice of architecture may be influenced by the kinds of analysis you require.

4. Refinements of Styles

The classification scheme of Section 3 maps out the space of architectures, but it does not capture all the richness found in nature. Each row can be elaborated to capture more detailed distinctions. These distinctions may matter because they determine whether pre-existing parts can be used together, or because they affect system performance or other system-level behavioral quantities.

In this section we expand Table 1 by partially elaborating two well-known families of styles that have been extensively analyzed by others. We do so with the intent of validating the classification: If the feature categories capture distinctions observed by others and the table extends smoothly, this increases our confidence in their relevance.

4.1 Pipe-and-Filter Systems and Dataflow networks

Dataflow networks describe systems whose components operate on large, continuously-available data stream. The components are organized in arbitrary topologies that stream the data with non-transforming connectors. What happens when restrictions are placed on the topologies?

Abowd, Allen, and Garlan analyzed what they call the *pipe and filter* style by way of formalizing the semantics of architectural styles (as opposed to cataloguing their construction, as we have done). They identify three (overlapping) variations of the pipe and filter style [A+95]:

- systems without feedback loops or cycles (acyclic)
- pipelines (linear), and
- systems with only fan-out components.

Their overall pipe and filter style corresponds to the dataflow network style of Table 1: the components are elements that asynchronously transform input into output with minimal retained state—i.e., transducers. The transducers are connected in various topologies by high-volume data flow streams.

The pipeline sub-style of [A+95] can be seen in Table 2 to be a specialization of dataflow network—its data and control topology is restricted from “arbitrary” in the general form to “linear” in the specialized form but the classifications are otherwise identical. The fan-out and acyclic sub-styles of [A+95] similarly differ from the general form only by imposing different topological restrictions.

Unix pipes and filters, a specialization not treated in [A+95] but widely used elsewhere, can be seen to be a sub-specialization of the pipeline style. The hook-ups can only be specified at the time a program—script, in this case—is written, or when the command is given to the operating system. Further, its components are those that accept ascii streams, not generalized data streams.

The classification shows that all of these styles (acyclic, pipelines, fan-out, Unix pipes and fil-

ters) indeed comprise a family that we have called “dataflow network”, in which the members are distinguished mainly by topological restriction. Table 2 shows the relationships among the major style and its family members.

Table 1: Specializations of the dataflow network style

<table>
<tr><th rowspan="2">Style</th><th colspan="2">Constituent parts</th><th colspan="3">Control issues</th><th colspan="4">Data issues</th><th colspan="2">Ctrl/data interaction</th></tr>
<tr><th>Components</th><th>Connectors</th><th>Topology</th><th>Synchronicity</th><th>Binding time</th><th>Topology</th><th>Continuity</th><th>Mode</th><th>Binding time</th><th>Isomorphic shapes</th><th>Flow directions</th></tr>
<tr><td colspan="12">Data flow styles: Styles dominated by motion of data through the system, with no “upstream” content control by recipient</td></tr>
<tr><td>Dataflow network [B+88]</td><td rowspan="5">transducers</td><td rowspan="4">data stream</td><td>arbitrary</td><td rowspan="5">asynch</td><td rowspan="4">i, r</td><td>arbitrary</td><td rowspan="5">cont lvol or hvol</td><td rowspan="5">passed</td><td rowspan="4">i, r</td><td rowspan="5">yes</td><td rowspan="5">same</td></tr>
<tr><td>• Acyclic [A+95]</td><td>acyclic</td><td>acyclic</td></tr>
<tr><td>• Fanout [A+95]</td><td>hierarchy</td><td>hierarchy</td></tr>
<tr><td>• Pipeline [DG90, Se88, A+95]</td><td rowspan="2">linear</td><td rowspan="2">linear</td></tr>
<tr><td>-Unix pipes and filters [Ba86a]</td><td>ascii stream</td><td>i</td><td>i</td></tr>
<tr><td colspan="12">Key to column entries</td></tr>
<tr><td>Synchronicity
Binding time
Continuity</td><td colspan="11">asynch (asynchronous)
i (invocation-time), r (run-time)
cont (continuous), hvol (high-volume), lvol (low-volume)</td></tr>
</table>

4.2 Cooperative Message-Passing Processes

Andrews [An91] analyzed and catalogued a family of styles based on processes communicating with each other via message-passing. This family corresponds to the communicating processes (CP) style in Table 1. Andrews identifies eight variants. The next sections show how each variant is a specialization of the basic CP style.

One-way data flow through networks of filters. This is a version of the dataflow network substyle described in Section 4.1 implemented with communicating processes; in this version the implementation with messages intrudes on the data flow abstraction. A piece of data enters the system and makes its way through a series of transformations, each transform accomplished by a separate process. The series need not be linear; Andrews gives an example of a tree of processes forming a sorting network. To analyze this sub-style, we note how it differs from both main styles that it resembles. To cast it as a specialization of dataflow networks, we (a) restrict its data and control topologies to one-way flows, and (b) relax its data-handling requirements from continuous to sporadic. To cast it as a specialization of communicating processes we restrict its topologies from arbitrary to one-way and its synchronicity to asynchronous. That is, this style lies within two subspaces of our design space.

Requests and replies between clients and servers. Clients and servers, a popular style, already occurs in Table 1. It can already be seen to be a specialization of the CP style in which the topologies, synchronicity, and mode are restricted from the general form. This is the naive form, which ignores the usual requirement to maintain state for an ongoing sequence of interactions between the client and the server.

Back-and-forth (heartbeat) interaction between neighboring processes. A heartbeat algorithm causes each node in the process graph to send information out (expand), and then gather in new information (contract). An example of applying this algorithm is to discover the topology of a network. On each "beat", each process (representing a processor) communicates with everyone it can, broadcasting its idea of the topology. Between beats, every process assimilates the information just sent to it, combining it with its current idea of the layout. The computation terminates when a completion condition has been met. Andrews proposes two variations of this sub-style, depending upon whether or not shared memory is used. We model this form of process interaction by restricting the synchronicity of the CP style to lock-step-parallel (although asynchronous versions are possible) and reflecting the shared-data/-distributed-data choice by describing the data and control topologies appropriately.

Probes and echoes in graphs. Probe/echo computations work on (incomplete) graphs. A probe is a message sent by a process to a set of successors; an echo is the reply. Probe/echo algorithms can be used to compute a depth-first search on a graph, discover network topologies, or broadcast using neighbors. Specializing the CP style by restricting the topologies to an incomplete graph, synchronicity to asynchronous, data mode to passed, and flow directions to same describes the probe/echo sub-style.

Broadcasts between processes in complete graphs. Broadcast algorithms use a distinguished process to send a message to all other processes. An example is to broadcast the value of a central clock in a soft-real-time system. Modelling the broadcast style in our classification simply requires restricting the data topology to star (for that portion of the computation involved in the broadcast) and the data mode to broadcast; the control topology remains arbitrary.

Token passing along edges in a graph. Token-passing algorithms use tokens (a special kind of message) to convey temporal rights to the processes receiving the tokens. Token-passing is used, for instance, in algorithms to compute the global state of a distributed asynchronous system, or to implement distributed mutual exclusion of a shared resource. Token-passing is a refinement of the CP style that restricts the synchronicity to asynchronous, data mode to passed, and flow direction to same. The topologies remain arbitrary, and the continuity remains sporadic low-volume.

Coordination between decentralized server processes. In this model, identical servers are replicated to increase the availability of services (for example, in case of the failure or backlog of a single server). The essence of the algorithm is to provide the appearance to clients of a single, centralized server; this requires that the servers coordinate with each other to maintain a consistent state. One server cannot change the "mutual" state without agreement of a sufficient majority of the others. This weighted voting scheme is implemented by passing multiple tokens among the servers. Architecturally, this algorithm is identical to the token-passing sub-style discussed above.

Replicated workers sharing a bag of tasks. Unlike decentralized servers that maintain multiple copies of data, this style provides multiple copies of computational elements. The replicated-workers style is a primary tool for single-instruction, multiple-data (SIMD) machine programmers. Parallel divide-and-conquer is one of its manifestations. One process can be the administrator, generating the first problem and assigning sub-problems. Other processes are workers, solving the sub-problems (and generating and administering further sub-problems as necessary). Sub-solutions bubble back up a hierarchical path until the original administrator can assemble the solution to the global problem. To see SIMD algorithms as a

sub-style of CP, we restrict the topologies to hierarchical, the synchronicity to synchronous, mode to passed or shared (depending on whether or not shared data is used) and flow direction to same.

Table 3 summarizes these descriptions.

Table 2: Specializations of the interacting processes style

Style	Constituent parts		Control issues			Data issues				Ctrl/data interaction	
	Components	Connectors	Topology	Synchronicity	Binding time	Topology	Continuity	Mode	Binding time	Isomorphic shapes	Flow directions
Interacting process styles: Styles dominated by communication patterns among independent, usually concurrent, processes											
Communicating processes [An91, Pa85]	processes	message protocols	arb	any but seq	w, c, r	arb	spor lvol	any	w, c, r	possibly	if isomorphic either
One-way data flow, networks of filters			linear	asynch		linear		passed	w, c	yes	same
Client/server request/reply			star	synch		star		passed		yes	opposite
Heartbeat			hier	ls/par		hier or star		passed shared ci/co		no	same
Probe/echo			incomplete graph	asynch		incomplete graph		passed		yes	same
Broadcast			arb	asynch		star		bdcast		no	same
Token passing			arb	asynch		arb.		passed		yes	same
Decentralized servers											
Replicated workers			hier	synch		hier		passed shared		yes	yes
Key to column entries											
Topology	hier (hierarchical), arb (arbitrary), star, linear (one-way)										
Synchronicity	seq (sequential, one thread of control), ls/par (lockstep parallel), synch (synchronous), asynch (asynchronous), opp (opportunistic)										
Binding time	w (write-time--that is, in source code), c (compile-time), i (invocation-time), r (run-time)										
Continuity	spor (sporadic), lvol (low-volume)										
Mode	shared, passed, bdcast (broadcast), mcast (multicast), ci/co (copy-in/copy-out)										

5. Using Styles in System Design

5.1 Supporting Styles in Architectural Design Languages

Specific styles are supported by a variety of frameworks and architectural description languages (ADLs) [Ga95]. For instance, every ADL in one survey was able to express a pipe-and-filter style, though few provided it as a built-in primitive [Cl96]. Some ADLs, however, go beyond that to support a diverse and open-ended (architect-defined) collection of styles. Two languages do this: Aesop [G+94] and UniCon [S+95].

Aesop is an object-oriented notation and system for developing style-specific architectural development environments. Its basic elements of architectural description are components,

connectors, and configurations. Styles are created by subtyping; style-specific vocabularies of design elements are created by defining the desired component types as subtypes of the generic component, the desired connector types as subtypes of the generic connector, and so on. Configuration rules are captured as part of the subtype definitions.

UniCon also defines components and connectors as the basic elements. Here, however, the language provides a collection of specific component and connector types. The set is manually extensible, and specializations are defined via property lists. Styles will be defined as restrictions on the available vocabulary.

5.2 Choosing Styles to fit the Problem

We expect that the distinctions established in this classification provide a framework for offering design guidance of the general form, "If your problem has characteristic X, consider architectures with characteristic Y". This form of design guidance was explored for the user interface component of systems by Lane [La90]. Lane characterized both his requirement and implementation domains as design spaces. He cast his design guidance as rules that mapped points in the requirement space to points in the implementation space with variable, signed weights. He wrote similar rules to describe compatibility of alternatives in the implementation space. This provided the ability to recommend candidate implementations for given problems.

The choice of an architectural style to fit a requirement is a similar task. We expect that an approach based on Lane's will be fruitful. However, organizing this information is a major undertaking for each domain. In the interim, we can at least state rules of thumb. Some of these are stated explicitly in analyses of architectural styles by cited authors; others are observations that derive directly from our classification.

- If your problem can be decomposed into sequential stages, consider batch sequential or pipeline architectures.
    - If in addition each stage is incremental, so that later stages can begin before earlier stages finish, consider a pipeline architecture.
- If your problem involves transformations on continuous streams of data (or on very long streams), consider a pipeline architecture.
    - However, if your problem involves passing rich data representations, avoid pipelines restricted to ASCII.
- If a central issue is understanding the data of the application, its management, and its representation, consider a repository or abstract data type architecture. If the data is long-lived, focus on repositories.
    - If the representation of data is likely to change over the lifetime of the program, then abstract data types can confine the change to particular components.
    - If you are considering repositories and the input data is noisy (low signal-to-noise ratio) and the execution order cannot be predetermined, consider a blackboard [Ni86]
    - If you are considering repositories and the execution order is determined by a stream of incoming requests and the data is highly structured, consider a database management system.
- If your system involves controlling continuing action, is embedded in a physical sys-

tem, and is subject to unpredictable external perturbation so that preset algorithms go awry, consider a closed loop control architecture [Sh95].

- If you have designed a computation but have no machine on which you can execute it, consider an interpreter architecture.
- If your task requires a high degree of flexibility/configurability, loose coupling between tasks, and reactive tasks, consider interacting processes.
    - If you have reason not to bind the recipients of signals from their originators, consider an event architecture.
    - If the tasks are of a hierarchical nature, consider a replicated worker or heartbeat style.
    - If the tasks are divided between producers and consumers, consider client/server.
    - If it makes sense for all of the tasks to communicate with each other in a fully connected graph, consider a token passing style.

6. Conclusion

Architectural styles are becoming the *lingua franca* of architecture-level design, in the same way that design patterns are moving to center stage in establishing the vocabulary and defining the solution space for finer-grained design problems. In order to capitalize on the shared experience represented by the repeated use of styles by system builders, it is necessary to establish a common vocabulary and a common descriptive framework for communicating about styles and the circumstances in which they are useful.

This paper provides such a framework. Based on the components and connectors that populate a style and how control and data are handled throughout the style, the framework serves as a set of distinguishing features for styles. Finer-grained distinctions may be made, leading to the notion of style families. We have shown how the framework accommodates two commonly-cited styles, communicating processes and dataflow networks. Finally, we have suggested how the framework can be used as the basis for imparting design guidance for choosing and using styles, by identifying those features that dominate the problem at hand.

The way now open for future work, which ranges from refinements of this classification and finer-grained characterizations of these and other styles, to building style-based architecture-level design environments that include analytical tools appropriate for each style.

7. Acknowledgments

The style classification and collection of rules of thumb have been developed over a period of several years. We particularly appreciate the ongoing involvement of David Garlan, Rob DeLine, and Greg Zelesnik in the discussion. Many others have offered useful advice, particularly our colleagues in the Composable Systems research group, members of the Software Architecture Reading Group and the students in Garlan and Shaw's software architecture course.

This work has profited from efforts of the Software Engineering Institute's Software Architecture Analysis Method (SAAM) group, especially Len Bass, Gregory Abowd, and Rick Kazman, who have provided a taxonomic classification of architectural elements (components and connectors), which provides the conceptual continuation of the work presented here.

This work has been supported by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency, under grant F33615-93-1-1330 and by a grant from Siemens Corporation. It represents the views of the author and not of Carnegie Mellon University or any of the sponsoring institutions The Software Engineering Institute is supported by the U. S. Department of Defense.

8. Bibliography

[A+95] Gregory D. Abowd, Robert Allen, David Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319-364, October 1995.

[AG94] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proc 16th International Conference on Software Engineering*, 1994.

[An91] Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1):49-90, March 1991.

[B+88] M. R. Barbacci, C. B. Weinstock, and J. M. Wing. Programming at the Processor-Memory-Switch Level. *Proc 10th Int'l Conf on Software Engineering*, April 1988.

[Ba86a] M. J. Bach. *The Design of the UNIX Operating System*. Software Series, Prentice-Hall 1986, sec 5.12, pp. 111-119.

[Ba86b] Robert M. Balzer. Living with the Next Generation Operating System. *Proc 4th World Computer Conf.*, September 1986.

[Be90] Laurence J. Best. *Application Architecture: Modern Large-Scale Information Processing*. Wiley, 1990.

[Bo86] Grady Booch. Object-Oriented Development. *IEEE Tr. Software Engineering*, February 1986, pp. 211-221.

[Br88] Frederick P. Brooks, Jr. Grasping Reality Through Illusion -- Interactive Graphics Serving Science. *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference*, May 1988, pp. 1-11.

[Cl96] Paul Clements. A Survey of Architecture Description Languages. *Proc International Workshop on Software Specification and Design*, Germany, 1996.

[CS95] James Coplien & Eric Schmidt (eds), *Pattern Languages of Program Design*, Addison-Wesley 1995.

[DG90] Norman Delisle and David Garlan. Applying Formal Specification to Industrial Problems: A Specification of an Oscilloscope. *IEEE Software*, September 1990.

[Fr85] Marek Fridrich and William Older. Helix: The Architecture of the XMS Distributed File System. *IEEE Software*, vol 2, no 3, May 1985 (pp.21-29).

[G+92] David Garlan, Gail Kaiser, and David Notkin. Using Tool Abstraction to Compose Systems. *IEEE Computer*, 25(6), June 1992.

[G+94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments. *Proc. Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, December 1994.

[Ga95] David Garlan (ed). First International Workshop on Architectures for Software Systems, Workshop Summary. *ACM Software Engineering Notes* 20(3), July 1995, pp.84-89.

[Ge89] C. Gerety. HP Softbench: A New Generation of Software Development Tools. *TR SESD-89-25*, Hewlett-Packard Software Engineering System Division, Ft. Collins CO, November 1989.

[GoF95] E. Gamma, R. Helm. R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1995.

[GS93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In Ambriola & Tortora (eds), *Advances in Software Engineering & Knowledge Engineering*, vol. II, World Scientific Pub Co., 1993, pp.1-39.

[HN86] Nico Habermann and David Notkin. Gandalf: Software Development Environments. *IEEE Tr on Software Engineering*, vol SE-12, December 1986.

[He69] Carl Hewitt. Planner A Language for Proving Theorems in Robots. *Proc First Int'l Joint Conf. in Artificial Intelligence*, 1969.

[HR85] Frederick Hayes-Roth. Rule-Based Systems. *Communications of the ACM*, vol 28, no 9, September 1985, pp.921-932.

[Ja96] Michael Jackson.Software *Requirements and Specifications: A Lexicon of Practice, Principles, and Prejudices*. Addison-Wesley 1995.

[K+96] Rick Kazman, Paul Clements, Gregory Abowd, Len Bass. Classifying Architectural Elements. *Proc ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996, submitted.

[KP88] 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, August / September 1988.

[La90] Thomas G. Lane. *User Interface Software Structures*. Ph.D. Thesis, Carnegie Mellon University, Carnegie Mellon University Computer Science Technical Report CMU-CS-90-101, June 1987.

[LS79] Hugh C. Lauer and Ed. H. Satterthwaite. Impact of MESA on System Design. *Proc Third Int'l Conf. on Software Engineering*, May 1979.

[Ni86] H. Penny Nii. Blackboard Systems. *AI Magazine* 7(3):38-53 and 7(4):82-107.

[Pa72] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. ACM* vol 15, December 1972.

[Pa85] Mark C. Paulk. The ARC Network: A Case Study. *IEEE Software*, vol 2 no 3, May 1985, pp. 62-69

[PN86] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4), November 1986, pp. 307-334.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, Oct 1992.

[Re90] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57-66, July 1990.

[S+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, May 1995.

[Se88] V. Seshadri et al. Semantic Analysis in a Concurrent Compiler. *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall 1996.

[Sh81] Mary Shaw (ed). *Alphard: Form and Content*. Springer-Verlag, 1981.

[Sh95] Mary Shaw. Beyond Objects: A Software Design Paradigm Based on Process Control. *ACM Software Engineering Notes*, 20(1), Jan 1995.

[Sh96] Mary Shaw. Some Patterns for Software Architectures. *Proceedings of Second Workshop on Pattern Languages for Programming*, Addison-Wesley 1996.

[Sp87] Alfred Z. Spector et al. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. *Carnegie Mellon University Computer Science Technical Report*, June 1987.