Formulations and Formalisms in Software Architecture*

Mary Shaw and David Garlan

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Abstract. Software architecture is the level of software design that addresses the overall structure and properties of software systems. It provides a focus for certain aspects of design and development that are not appropriately addressed within the constituent modules. Architectural design depends heavily on accurate specifications of subsystems and their interactions. These specifications must cover a wide variety of properties, so the specification notations and associated methods must be selected or developed to match the properties of interest. Unfortunately, the available formal methods are only a partial match for architectural needs, which entail description of structure, packaging, environmental assumptions, representation, and performance as well as functionality. A prerequisite for devising or selecting a formal method is sound understanding of what needs to be formalized. For software architecture, much of this understanding is arising through *progressive codification*, which begins with real-world examples and creates progressively more precise models that eventually support formalization. This paper explores the progressive codification of software architecture: the relation between emerging models and the selection, development, and use of formal systems.

1 Status and Needs of Software Architecture

As software systems become more complex, a critical aspect of system design is the overall structure of the software and the ways in which that structure provides conceptual integrity for the system. This level of system design has come to be known as *software architecture* [GS93, PW92].

In an architectural design, systems are typically viewed as compositions of module-scale, interacting components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions between components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database accessing protocols, asynchronous event multicast, and piped streams.

While it has long been recognized that finding an appropriate architectural design for a system is a key element of its long-term success, current practice for describing architectures is typically informal and idiosyncratic. Usually, architectures are represented abstractly as box and line diagrams, together with accompanying prose

* To appear in *Volume 1000*, Springer-Verlag Lecture Notes in Computer Science, 1995.

that explains the meanings behind the symbols, and provides some rationale for the specific choice of components and interactions.

The relative informality and high level of abstraction of current practice in describing architectures might at first glance suggest that architectural descriptions have little substantive value for software engineers. But there are two reasons why this is not the case. First, over time engineers have evolved a collection of idioms, patterns, and styles of software system organization that serve as a shared, semantically-rich vocabulary between engineers. For example, by identifying a system as an instance of a pipe-filter architectural style an engineer communicates the facts that the system is primarily involved in stream transformation, that the functional behavior of the system can be derived compositionally from the behaviors of the constituent filters, and that issues of system latency and throughput can be addressed in relatively straightforward ways. Thus, although this shared vocabulary is largely informal, it conveys considerable semantic content between software engineers.

The second reason is that although architectural structures may themselves abstract away from details of the actual computations of the elements, those structures provide a natural framework for understanding broader system-level concerns, such as global rates of flow, patterns of communication, execution control structure, scalability, and intended paths of system evolution. Thus, architectural descriptions serve as a skeleton around which system properties can be fleshed out, and thereby serve a vital role in exposing the ability of a system to meet its gross system requirements.

This is, of course, not to say that more formal notations for architectural description and rigorous techniques of analysis are unnecessary. Indeed, it is clear that much could be gained if the current practice of architectural design could be supported with better notations, theories, and analytical techniques. In this paper we explore specification issues in software architecture, with particular attention to the way improvements in the formulation of architectural issues sets the stage for better formalization.

1.1 Current Status

Over the past few years, recognition of the significance of software architecture has led to considerable research and development activity, both in industry and academia. These activities can be roughly placed into four categories.

The first category is addressing the problem of architectural characterization by providing new *architectural description languages*. As detailed later, these languages are aimed at giving practitioners better ways of writing down architectures so that they can be communicated to others, and in many cases analyzed with tools.

The second category is addressing *codification of architectural expertise* [GHJV94, GS93]. Work in this area is concerned with cataloging and rationalizing the variety of architectural principles and patterns that engineers have developed through software practice.

The third category is addressing *frameworks for specific domains* [DAR90, Tra94]. This work typically results in an architectural framework for a specific class of software such as avionics control systems, mobile robotics, or user interfaces. When successful, such frameworks can be easily instantiated to produce new products in the domain.

The fourth category addresses *formal underpinnings for architecture*. As new notations are developed, and as the practice of architectural design is better understood, formalisms for reasoning about architectural designs become relevant. Several of these are described later.

1.2 What Needs to be Specified about Architectures

Architectural design determines how to compose systems from smaller parts so the result meets system requirements. Most system requirements extend beyond functionality to a variety of other properties that matter to the client. Moreover, the correctness of the composition depends at least as much on the component interactions and on the assumptions components make about their execution environment as it does on what the components actually compute.

Accordingly, architectural specifications must address the extra-functional properties of components (structure, packaging, environmental dependencies, representation, and performance), the nature of the interactions among components, and the structural characteristics of the configurations.

Structural Properties. The most significant properties for architectural design deal with the ways components interact, and hence with the ways those components can be combined into systems. The packaging of a component includes the type of component and the types of interactions it is prepared to support. The choice of packaging is often largely independent of the underlying functionality, but components must be packaged in compatible ways if they are to work together smoothly. For example, Unix provides both a sort system call and a sort filter; while they have the same functionality, they are far from interchangeable.

Some common packagings for components and the ways they interact are:

Component type	Common types of interaction
Module	Procedure call, data sharing
Object	Method invocation (dynamically bound procedure call)
Filter	Data flow
Process	Message passing, remote procedure call various communication protocols, synchronization
Data file	Read, write
Database	Schema, query language
Document	Shared representation assumptions

Distinctions of this kind are now made informally, often implicitly. If the distinctions were more precise and more explicit, it would be easier to detect and eventually correct incompatibilities by analyzing the system configuration description. Such checking must address not only local compatibility (e.g., do two components expect the same kinds of interactions), but also global properties (e.g., no loops in a data flow system).

Extra-functional Properties. In addition to functionality and packaging, architectural specifications must be capable of expressing extra-functional properties related to performance, capacity, environmental assumptions, and global properties such as reliability and security [Sha85, MCN92, CBKA95]. Many of these additional properties are qualitative, so they may require different kinds of support from more formal specifications. These other properties include:

time requirements	ease of use
timing variability	reliability
real-time response	robustness
latency	service capacity (e.g., # of clients/server)
throughput	possession of main thread of control
bandwidth	dependence on specific libraries, services
space requirements	conformance to an interface standard
space variability	conformance to implementation standard
adaptability	intended profile of operation usage
precision and accuracy	minimum hardware configuration
security	need to access specialized hardware

For example, this product description specifies the interface between a software product and the operating system/hardware it requires [Com95].

1. IBM or 100% IBM-compatible microcomputer with Intel 80386 microprocessor or higher or 100%-compatible processor.
2. Minimum 4 MB RAM.
3. 3 MB of available space on a hard disk.
4. ISO 9660-compatible CD-ROM drive with 640+ MB read capacity and Microsoft CD-ROM extensions.
5. Microsoft Windows'-compatible printer (not plotter) recommended, with 1.5 MB printer memory for 300 dpi laser printing, 6 MB for 600 dpi.
6. Microsoft Windows'-compatible mouse (recommended).
7. Microsoft Windows'-compatible VGA card and monitor.
8. Microsoft Windows' version 3.1 and MS-DOS version 4.01 or later.

This specification deals with space and with conformance to established standards. The functionality of the product is described (imprecisely) in associated prose and pictures.

Families of Related Systems. In addition to structure and packaging, architectural specifications must also deal with families of related systems. Two important classes of system family problems are:

1. Architectural styles that describe families of systems that use the same types of components, types of interactions, structural constraints, and analyses. Systems built within a single style can be expected to be more compatible than those that mix styles: it may be easier to make them interoperate, and it may be easier to reuse parts within the family.

2. Some systems can accommodate a certain amount of variability: they depend critically on some central essential semantics, and they require certain other support to be present, but do not rely on details. In operating systems, these are sometimes distinguished as policy and mechanism, respectively: for example, it's important for a synchronization mechanism to prevent interference, deadlock, and starvation, but the details of process ordering are incidental.

2 Models and Notations for Software Architectures

Software systems have always had architectures; current research is concerned with making them explicit, well-formed, and maintainable. This section elaborates on the current practice, describes the models that are now emerging and the languages that support some of those models, and discusses the standards that should be used to evaluate new models and tools in this area.

2.1 Folklore and Common Practice

As noted above, software designers describe overall system architectures using a rich vocabulary of abstractions. Although the descriptions and the underlying vocabulary are imprecise and informal, designers nevertheless communicate with some success. They depict the architectural abstractions both in pictures and words.

"Box-and-line" diagrams often illustrate system structure. These diagrams use different shapes to suggest structural differences among the components, but they make little discrimination among the lines—that is, among different kinds of interactions. The architectural diagrams are often highly specific to the systems they describe, especially in the labeling of components. For the most part, no rules govern the diagrams; they appeal to rich intuitions of the community of developers. Diagramming rules do exist for a few specific styles—data flow diagrams and some object-oriented disciplines, for example.

The diagrams are supported by prose descriptions. This prose uses terms with common, if informal, definitions (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." [S$^+$87]
- "*Abstraction layering* and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a *client-server model* for the structuring of applications." [FO85]
- "We have chosen a *distributed, object-oriented approach* to managing information." [Lin87]
- "The easiest way to make the canonical sequential compiler into a concurrent compiler is to *pipeline* the execution of the compiler phases over a number of processors. ... A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program." [S$^+$88]

– "The ARC network [follows] the *general network architecture* specified by the ISO in the Open Systems Interconnection Reference Model. It consists of physical and data layers, a network layer, and transport, session, and presentation layers." [Pau85]

We studied sets of such descriptions and found a number of abstractions that govern the overall organization of the components and their interactions [GS93]. A few of the patterns, or styles, (e.g., object organizations [Boo86] and blackboards [Nii86]) have been carefully refined, but others are still used quite informally, even unconsciously. Nevertheless, the architectural patterns are widely recognized. System designs often appeal to several of these patterns, combining them in various ways.

2.2 Emerging Models

Most of the current work on software architecture incorporates models, either explicit or implicit, of the conceptual basis for software architecture. Some of this work is directed at refining models; other work implicitly adopts a model in pursuit of some other goal. Five general types of models appear with some regularity: structural, framework, dynamic, process, and functional. Of these, structural and dynamic models are most common. The representative examples here were discussed at the First International Workshop on Architectures for Software Systems [Gar95] and the Dagstuhl Workshop on Software Architecture [GPT95].

Structural Models. The most common model views architecture as primarily *structural*. This family of models shares the view that architecture is based on components, connectors, and "other stuff". The "other stuff" in various ways reaches beyond structure to capture important semantics. Although there is not as yet consensus on precisely what that semantics is, "other stuff" includes configuration, rationale, semantics, constraints, style, properties, analyses, and requirements or needs. As detailed later, structural models are often supported by architecture description languages. Examples include Aesop, C2, Darwin, UniCon, and Wright. A shared interchange language, ACME, is being developed.

Framework Models. A second group of models is similar to structural models, but places more emphasis on the coherent structure of the whole than on describing structural details. These *framework* models often focus on one specific structure, for example, one that targets a specific class of problems or market domain. Narrowing the focus permits a richer elaboration for the domain of interest. Examples include various domain-specific software architectures (DSSAs), MetaObject Protocols (MOPs), CORBA and other object interaction models, component repositories, and SBIS.

Dynamic Models. *Dynamic* models are complementary to structural or framework models. They address large-grain behavioral properties of systems, often describing reconfiguration or evolution of the systems. "Dynamic" may refer to changes in the overall system configuration, to setting up and taking down pre-enabled paths, or to the progress of computation (changing data values, following a control thread). These systems are often reactive. Examples include the Chemical Abstract Machine, Archetype, Rex, Conic, Darwin, and Rapide.

Process Models. Another, smaller, family, the process models, is constructive, operational, and imperative. These models focus on the construction steps or processes that yield a system. The architecture is then the result of following some process script. Examples include some aspects of Conic and process programming for architecture.

Functional Models. A minority regards architecture as a set of *functional* components, organized in layers that provide services upward. It is perhaps most helpful to think of this as a particular framework.

2.3 Architectural Description Languages

Of the models described in Section 2.2, the structural models are now most prevalent. A number of architecture description languages (ADLs) are being developed to support these models. ADLs typically support the description of systems in terms of typed *components* and sometimes *connectors* that make the abstractions for interaction first-class entities in the language. They often provide a graphical interface so that developers can express architectures using diagrams of the kind that have proven useful. Current ADLs include Aesop [GAO94], ArTek [T+94], Darwin [MK95], Rapide [LAK+95], UniCon [SDK+95], and Wright [AG94].

While all of these ADLs are concerned with architectural structure, they differ in their level of *genericity*. There are three basic levels. Some are primarily concerned with *architectural instances*. That is, they are designed to describe specific systems, and provide notations to answer the questions of the form "What is the architecture of system S?" ArTek, Rapide, and UniCon are in this category.

Other ADLs are primarily concerned with *architectural style*. That is, they are designed to describe patterns, or idioms, of architectural structure. The notations in this category therefore describe families of systems, and answer questions of the form "What organizational patterns are are used in system S?", or "What is the meaning of architectural style T?" Aesop is in this category.

ADLs associated with styles typically attempt to capture one or more of four aspects of style: the underlying intuition behind the style, or the system model; the kinds of components that are used in developing a system according to the pattern; the connectors, or kinds of interactions among the components; and the control structure or execution discipline.

Still other ADLs are concerned with *architecture in general*. They attempt to give meaning to the broader issues of the nature of architecture, and the ways architectural abstractions can provide analytic leverage for system design. Several of these are considered later in this paper.

2.4 Evaluation Criteria

Languages, models, and formalisms can be evaluated in a number of different ways. In this case, the models and the detailed specifications of relevant properties have a utilitarian function, so appropriate evaluation criteria should reflect the needs of software developers. These criteria differ from the criteria used to evaluate formalisms for mathematical elegance.

Expertise in any field requires not only higher-order reasoning skills, but also a large store of facts, together with a certain amount of context about their implications and appropriate use. This is true across a wide range of problem domains; studies have demonstrated it for medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others [Red88, Sim87]. An expert in a field must know around 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived. Chunks are typically operational: "in *this* situation, do *that*". Furthermore, full-time professionals take ten years to reach world-class proficiency. It follows that models and tools intended to support experts should support rich bodies of operational knowledge. Further, they should support large vocabularies of established knowledge as well as the theoretical base for deriving information of interest.

Contrast this with the criteria against which mathematical systems are evaluated. Mathematics values elegance and minimality of mechanism; derived results are favored over added content because they are correct and consistent by their construction.

Architecture description languages are being developed to make software designers more effective. They should be evaluated against the utilitarian standard, preferring richness of content and relevance to the application over elegance and minimality. This implies, for example, that these languages should support— directly—the breadth of architectural abstractions that software designers use: data flow (including pipes and filters), object-oriented, functional, state-based, message-passing, and blackboard organizations. The fact that these abstractions could all be expressed using some one of the styles is interesting but not grounds for impeding the choice of abstractions relevant to the project at hand.

3 Progressive Codification

Software specification techniques have often evolved in parallel with our understanding of the phenomena that they specify.

This development can bee seen in the development of data types and type theory [Sha80]. In the early 1960s, type declarations were added to programming languages. Initially they were little more than comments to remind the programmer of the underlying machine representation. As compilers became able to perform syntactic validity checks the type declarations became more meaningful, but "specification" meant little more than "procedure header" until late in the decade. The early 1970s brought early work on abstract data types and the associated observation that their checkable redundancy provided a methodological advantage because they gave early warning of problems. At this time the purpose of types in programming languages was to enable a compile-time check that ensured that the actual parameters presented to a procedure at runtime would be acceptable. Through the 1980s type systems became richer, stimulated by the introduction of inheritance mechanisms. At the same time, theoretical computer scientists began developing rich theories to fully explain types. Now we see partial fusion of types-in-languages and types-as-theory in functional languages with type inference. We see in this history that theoretical elaboration relied on extensive experience with the phenomena, while at the same time practicing programmers are willing to write down specifications only to the extent that they are rewarded with analysis than simplifies their overall task.

Thus, as some aspect of software development comes to be better understood, more powerful specification mechanisms become available, and they yield better rewards for the specification effort invested. We can characterize some of the levels of specification power:

1. *Capture:* retain, explain, or retrieve a definition
2. *Construction:* explain how to build in instance from constituent parts
3. *Composition:* say how to join pieces (and their specifications) to get a new instance
4. *Selection:* guide designer's choice among implementation alternatives or designs
5. *Verification:* determine whether an implementation matches specification
6. *Analysis:* determine the implications of the specification
7. *Automation:* construct an instance from an external specification of properties

When describing, selecting, or designing a specification mechanism, either formal or informal, it is useful to be explicit about which level it supports. Failure to do so leads to mismatches between user expectations and specification power.

Architecture description languages provide a notation for *capturing* system descriptions. Several have associated tools that will *construct* instances from modules of some programming language. At least one technique for design *selection* has been developed [Lan90]. Support for other levels of aspiration is spotty.

No matter how badly we would like to leap directly to fully formal architectural specifications that support analysis and automation, history says we must first make our informal understanding explicit, then gradually make it more rigorous as it matures. In this way the specification mechanisms may be appropriate for the properties that they specify. Application of existing formal methods in inappropriate ways will fail to come to grips with the essential underlying problems.

4 Practice and Prospects for Formalisms

To illustrate the ways in which software architecture is being progressively codified, we now outline some of the formalisms that have been developed for software architecture. The goal here is not to provide a complete enumeration, but rather to indicate broadly the kinds of formalisms that are being investigated by the software architecture research community, and the extent to which those formalisms have been successful.

4.1 Formalisms in Use for Architecture

In order to make sense of the variety of existing formal approaches, it helps to have an organizational framework. One such framework positions architectural formalisms in two dimensional space that follows directly from the distinctions made in previous sections. Along one dimension is the *genericity* of the formalism. As outlined in Sect. 2.3, architectural concerns may relate to (a) a specific system (or architectural instance), (b) a family of systems (or architectural style), or (c) architecture in general. Along the second dimension is the *power* of the formalism. As outlined in Sect. 3, aspirations of different formal notations can vary from system documentation to automated system construction.

In practice, most formalisms address several aspects of this space. For example, a formal notation for architectural styles might be useful both for system construction as well as support verification. In most cases, however, a formalism has at its core a specific problem that it is attempting to address. It is this core that we are most interested in. Here are four core functions in this design space.

Analysis of Architectural Instances. Going beyond the base-level capability of architectural description languages to express specific system designs is the need to perform useful analyses of the designs. To do this requires the association of an underlying semantic model with those descriptions. Several such models have been proposed. These differ substantially depending on the kind of model they consider.

To take four representative examples:

1. Rapide models the behavior of a system in terms of partially ordered sets of events [LAK+95]. Components are assigned specifications that allow the system's event behavior to be simulated and then analyzed. Typical kinds of analyses reveal whether there is a causal dependency between certain kinds of computations. The presence (or absence) of these causality relationships can sometimes indicate errors in the architectural design.
2. Darwin [MK95] models system behavior in terms of the π-calculus [MPW92]. The flexibility of this model permits the encoding of highly dynamic architectures, while the strong typing system of the π-calculus permits certain static checks. For example, it can guarantee that processes only talk over channels of the correct type, even though the number and connectivity of those channels may change during runtime.
3. UniCon [SDK+95] and Aesop [GAO94] support methods for real-time analysis—RMA and EDF, respectively. These ADLs, and their supporting tools, capture relevant information, repackage it into the formats required by real-time analysis tools, which are then invoked to perform the analyses.
4. Wolf and Inverardi have explored the use of Chemical Abstract Machine notation [BB92] for modelling architectural designs [IW95]. This model also deals well with dynamic behavior of a system, essentially providing a kind of structural rewrite system.

Capture of Architectural Styles. When people refer to a system as being in a pipe-filter style it may not be clear precisely what they mean. While it may be clear that such a system should be decomposed into a graph of stream transformers, more detailed issues of semantics are typically left underspecified, or may vary from system to system. For example, are cycles allowed? Must a pipe have a single reader and a single writer? Can filters share global state?

In an attempt to provide more complete semantics for some specific styles a number of styles have been completely formalized. For example, Allen and Garlan [AG92] provide a formalization of a pipe-filter architectural style, while [GN91] develops a formalization of implicit-invocation architectural style. Both of these use the Z specification language [Spi89]

Generalizing from these examples, Abowd, Allen, and Garlan [AAG93] describe a denotational framework for developing formal models of architectural style (also in Z). The idea of the framework is that each style can be defined using three functions that indicate how the syntactic, structural aspects of the style are mapped into semantic entities. The authors argue that when several styles are specified in this way, it becomes possible to compare those styles at a semantic level. For example, is one style a substyle of another? Does a property of one style hold of another?

A somewhat different approach arises in the context of architectures for specific product families. A number of such "domain-specific" software architectures have been formalized. One of the more prominent is in the avionics domain [BV93]. Here a language, called Meta-H, was developed to capture the architectural commonality among the applications, and to provide high-level syntactic support for instantiating the framework for a specific product. The language was designed to reflect the vocabulary and notations that avionics control systems engineers (as opposed to software engineers) routinely used in their designs.

Verification of Architectural Styles. In many cases architectural descriptions are at a sufficiently abstract level that they must be refined into lower-level architectural descriptions. Typically, the lower-level descriptions are in terms of design entities that are more directly implemented than their abstract counterparts. For example, a system that is described at an abstract level as a dataflow system, might be recast in terms of shared variable communication at a lower (but still architectural) design level.

Moriconi and his colleagues have observed that it is possible to exploit patterns of refinement between different levels of architectural description [MQR95]. For instance, refining a dataflow connector to a shared variable connector involves a stylized transformation of asynchronous reading/writing to synchronized data access.

To capitalize on this observation they have proposed formalisms that encode transformations between architectural styles. In the above example, they might provide a pipe-to-shared-data transformation. The goal is to provide a complete enough system of transformations that machine aided refinement can take place. Moreover, as they note, by factoring out proofs of correctness of refinement at the style (or family) level, they simplify the work needed to carry out the refinement between any two architectural instances written in the respective styles.

Analysis of Architecture in General. When considering architectural design broadly, a number of formal questions arise: What does it mean to have a consistent or a complete architectural description? What is the formal nature of architectural connection?

The Wright specification language represents first steps toward answering these kinds of questions [AG94]. In this language, connectors are viewed as first class entities, defined as a set of protocols. Similarly, interfaces to components are described in terms of the protocols of interaction with their environment. Given this formal basis, it is possible to ask whether a given connector can be legally associated with a given component interface. This amounts to a test for protocol compatibility. Wright protocols are defined as CSP processes [Hoa85], and protocol compatibility can be reduced to a check of process refinement. The result of such a check is a strong guarantee that components interacting over a given connector will never deadlock.

4.2 What's Missing?

Standing back from the specific formalisms currently under development, two salient facts stand out.

First, consistent with the multi-faceted nature of software architecture itself, formalisms for modelling architecture are attempting to address a wide range of different issues. There is both good and bad news in this. The good news is that we are making incremental progress on understanding ways to lend precision and analytic capability to architectural design. The bad news is that the diversity of approaches leads to a fragmented and, in some cases, conflicting set of formal models for architecture. Consequently, no general unifying picture has emerged. And worse, we have no good ways of relating the various models to each other.

Second, existing formalisms address only a small portion of the needs of software architects. By and large, the formal approaches to software architecture have concentrated on the functional *behavior* of architectural descriptions. That is, they typically provide computational models, or ways of constructing them, that expose issues of data flows, control, sequencing, and communication. While useful and necessary, this is only a starting point. In addition—and arguably more important—are the extra-functional aspects of systems, such as performance, reliability, security, modifiability, and so on. Currently we do not know how to provide the necessary calculi for these other kinds of issues, or to relate these systems of reasoning to the existing formalisms for architecture.

5 Current Opportunities

Although the structure of software has been a concern for decades, software architecture has only recently emerged as an explicit focus of research and development. While considerable progress has been made over the last 5-10 years in recognizing the needs of practitioners for codifying, disseminating, describing, and analyzing architectural designs, there remain many, many open problems. Some of these problems might well be solved by better use of formalisms, provided they can bend to the needs of the practice (and not the other way around). Here are some areas that challenge current formalisms; they present promising research opportunities.

5.1 Heterogeneity

Practical systems are heterogeneous in structure and packaging. No matter how desirable it may be for a system to be composed entirely from a single family of compatible components, for most complex systems structural heterogeneity is inevitable. Strict adherence to a single style throughout a system is often impractical, so articulation between styles will be required. Furthermore, components with the desired functionality will often be packaged in different ways. Therefore, we need to find ways to handle packaging incompatibility [GAO95].

Heterogeneity arises from multiple packaging standards, from legacy systems that will not or cannot be rewritten, and from differences in usage within a single standard. At present, many ad hoc techniques are used to compensate for the incompatibility of parts. It would be useful to develop a systematic model that explains these and provides guidance for choosing the appropriate technique for a given situation [Sha95].

5.2 Incomplete and Evolving Specifications

According to conventional doctrine, component specifications are

1. sufficient (say everything you need to know)
2. complete (are the only source of information)

However, architectural elements violate this doctrine. Architectural needs are open-ended, and a designer cannot anticipate all the properties of interest. Specifications are incomplete and evolving. Moreover, gathering specification information incurs costs; even for common properties, completeness may be impractical, and the cost may be prohibitive for uncommon properties. Even worse, interesting properties may emerge after a component is released (e.g., "upward compatible with Fenestre version 4.5") [GAO95].

Notably, we often make progress with only minimal information. Sometime we can take advantage of new information when it comes along. A promising research opportunity is understanding how to make architectural specifications partial, incremental, and evolving. This entails adding new properties, declaring what properties are required for specific kinds of analysis, checking consistency, and propagating new information to improve old analyses. Work on using partial specifications will help [Jac94, Per87], as will a fresh approach that views them as evolving entities rather than static documents.

5.3 Extra-functional Properties

We have already noted the failure of most existing formalisms to handle properties that go beyond the computational behavior of the system. It will be a challenge to find formal systems for reasoning about the kinds of properties listed in 1.2 at an architectural level.

5.4 Multiple Views

Complex specifications require structure, such as different segments for different concerns. However, different concerns also lead to different notations. As indicated in Sect. 4.2, this leads to a multiple-view problem: different specifications describe different, but overlapping issues.

For example, formalisms that are good at accounting for dynamic properties of architectures may not be good for performing static/global analyses. For example, Wright (based on CSP) permits some powerful static checks of deadlock freedom, but does not deal with dynamic creation of processes. On the other hand, Darwin (based on the π-calculus) permits flexible description of dynamic architectures, but is less well-suited to proofs about absence of deadlock.

5.5 Classification and taxonomy

Software designers use a wide variety of styles, which are built up from identifiable types of components and interactions (or connectors). In practice, we see an enormous amount of variation on each of these themes. In order to be support checking and analysis, the specifications must be much more precise than at present. A classification or taxonomy for styles, components, and connectors would be a major step toward declaring and checking these structural types.

6 Research Support

This work reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency, under grant F33615-93-1-1330, by a grant from Siemens Corporation, by National Science Foundation Grant CCR-9357792, and by Federal Government Contract Number F19628-90-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a Federally Funded Research and Development Center. Some of this material was previously covered in position papers for the First Workshop on Pattern Languages for Programming and the First International Workshop on Architectures for Software Systems.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[AG92] Robert Allen and David Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proceedings of IFIP'92*. Elsevier Science Publishers B.V., September 1992.

[AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, (96):217–248, 1992.

[Boo86] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[BV93] Pam Binns and Steve Vestal. Formal real-time architecture specification and analysis. In *Tenth IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[CBKA95] Paul Clements, Len Bass, Rick Kazman, and Gregory Abowd. Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality*, Austin, Texas, October 1995.

[Com95] DeLorme Mapping Company. WWW page describing MapExpert product, 1995. URL: http://www.delorme.com/catalog/mex.htm.

[DAR90] *Proceedings of the Workshop on Domain-Specific Software Architectures*, Hidden Vallen, PA, July 1990. Software Engineering Institute.

[FO85] Marek Fridrich and William Older. Helix: The architecture of the XMS distributed file system. *IEEE Software*, 2(3):21–29, May 1985.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 179–185. ACM Press, December 1994.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, April 1995.

[Gar95] David Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995. Published as CMU Technical Report CMU-CS-95-151, April 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, October 1991. Springer-Verlag, LNCS 551.

[GPT95] David Garlan, Frances Newberry Paulisch, and Walter F. Tichy, editors. *Summary of the Dagstuhl Workshop on Software Architecture*, February 1995. Reprinted in ACM Software Engineering Notes, pages 63-83, July 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, Singapore, 1993. World Scientific Publishing Company. Also appears as SCS and SEI technical reports: CMU-CS-94-166, CMU/SEI-94-TR-21, ESC-TR-94-021.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[IW95] Paola Inverardi and Alex Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):373–386, April 1995.

[Jac94] Michael Jackson. Problems, methods and specialisation (a contribution to the special issue on software engineering in the year 2001). *IEE Software Engineering Journal*, November 1994. A shortened version of this paper also appears in IEEE Software, November 1994, 11(6).

[LAK+95] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):336–355, April 1995.

[Lan90] Thomas G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18 ESD-90-TR-219, Carnegie Mellon University, September 1990.

[Lin87] Mark A. Linton. Distributed management of a software database. *IEEE Software*, 4(6):70–76, November 1987.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6), June 1992.

[MK95] Jeff Magee and Jeff Kramer. Modelling distributed software architectures. In *Proceedings of the First International Workshop on Architectures for Software Systems*. Reissued as Carnegie Mellon University Technical Report CMU-CS-95-151, April 1995.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):356–372, April 1995.

[Nii86] H. Penny Nii. Blackboard systems Parts 1 & 2. *AI Magazine*, 7 nos 3 (pp. 38-53) and 4 (pp. 62-69), 1986.

[Pau85] Mark C. Paulk. The ARC Network: A case study. *IEEE Software*, 2(3):61–69, May 1985.

[Per87] Dewayne E. Perry. Software interconnection models. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 61–68, Monterey, CA, March 1987. IEEE Computer Society Press.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Red88] Raj Reddy. Foundations and grand challenges of artificial intelligence. *AI Magazine*, 9(4):9–21, Winter 1988. 1988 presidential address, AAAI.

[S$^+$87] Alfred Z. Spector et al. Camelot: A distributed transaction facility for Mach and the Internet—an interim report. Technical Report CMU-CS-87-129, Carnegie Mellon University, June 1987.

[S$^+$88] V. Seshadri et al. Semantic analysis in a concurrent compiler. In *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. ACM SIGPLAN Notices, 1988.

[SDK$^+$95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture*, 21(4):314–335, April 1995.

[Sha80] Mary Shaw. The impact of abstraction concerns on modern programming languages. In *Proceedings of the IEEE Special Issue on Software Engineering*, volume 68, pages 1119–1130, September 1980.

[Sha85] Mary Shaw. What can we specify? Questions in the domains of software specifications. In *Proceedings of the Third International Workshop on Software Specification and Design*, pages 214–215. IEEE Computer Society Press, August 1985.

[Sha95] Mary Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proceedings of the Symposium on Software Reuse*, April 1995.

[Sim87] Herbert A. Simon. Human experts and knowledge-based systems, November 9-12 1987.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[T+94] Allan Terry et al. Overview of teknowledge's domain-specific software architecture program. *ACM SIGSOFT Software Engineering Notes*, 19(4):68–76, October 1994.

[Tra94] Will Tracz. Collected overview reports from the DSSA project. Loral Federal Systems - Owego, October 1994.