Software Architecture: a Roadmap

David Garlan
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
(412) 268-5056
garlan@cs.cmu.edu

ABSTRACT
Over the past decade software architecture has received increasing attention as an important subfield of software engineering. During that time there has been considerable progress in developing the technological and methodological base for treating architectural design as an engineering discipline. However, much remains to be done to achieve that goal. Moreover, the changing face of technology raises a number of new challenges for software architecture. This paper examines some of the important trends of software architecture in research and practice, and speculates on the important emerging trends, challenges, and aspirations.

Keywords
Software architecture, software design, software engineering

1 INTRODUCTION

A critical issue in the design and construction of any complex software system is its architecture: that is, its gross organization as a collection of interacting components. A good architecture can help ensure that a system will satisfy key requirements in such areas as performance, reliability, portability, scalability, and interoperability. A bad architecture can be disastrous.

Over the past decade software architecture has received increasing attention as an important subfield of software engineering. Practitioners have come to realize that getting an architecture right is a critical success factor for system design and development. They have begun to recognize the value of making explicit architectural choices, and leveraging past architectural designs in the development of new products. Today there are numerous books on architectural design, regular conferences and workshops devoted specifically to software architecture, a growing number of commercial tools to aid in aspects of architectural design, courses in software architecture, major government and industrial research projects centered on software architecture and an increasing number of formal architectural standards. Codification of architectural principles, methods, and practices has begun to lead to repeatable processes of architectural design, criteria for making principled tradeoffs among architectures, and standards for documenting, reviewing, and implementing architectures.

In *The Future of Software Engineering*, A. Finkelstein, ed., ACM Press, 2000.

However, despite this progress, as engineering disciplines go, the field of software architecture remains relatively immature. While the outlines of an engineering basis for software architecture are becoming clear, there remain numerous challenges and unknowns. We can therefore expect to see major new developments in the field over the next decade – both in research and practice. Some of these developments will be natural extensions of the current trajectory. But there are also a number of radical new opportunities, brought about by the changing face of technology.

In this paper I examine some of the important trends of software architecture in research and practice. To set the stage, I begin by describing the roles of architecture in software systems development. Next I summarize the past and current state of research and practice. Finally, after considering some of the forces that are changing the world of software systems themselves, I speculate on emerging trends, challenges, and aspirations.

2 THE ROLES OF SOFTWARE ARCHITECTURE

While there are numerous definitions of software architecture, at the core of all of them is the notion that the architecture of a system describes its gross structure. This structure illuminates the top level design decisions, includeing things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high-level analysis and critical appraisal.

Software architecture typically plays a key role as a bridge between requirements and implementation (see Figure 1).

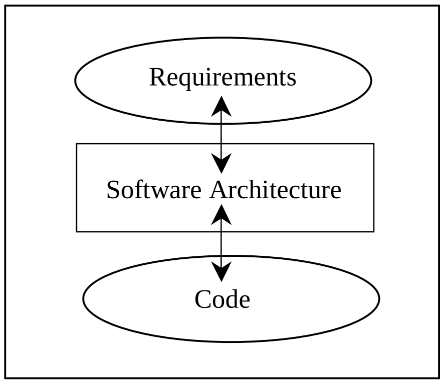

Figure 1: Software Architecture as a Bridge

By providing an abstract description of a system, the architecture exposes certain properties, while hiding others. Ideally this representation provides an intellectually tractable guide to the overall system, permits designers to reason about the ability of a system to satisfy certain requirements, and suggests a blueprint for system construction and composition. For example, an architecture for a signal processing application might be constructed as a dataflow network in which the nodes read input streams of data, transform that data, and write to output streams. Designers might use this decomposition, together with estimated values for input data flows, computation costs, and buffering capacities, to reason about possible bottlenecks, resource requirements, and schedulability of the computations.

To elaborate, software architecture can play an important role in at least six aspects of software development.

1. *Understanding:* Software architecture simplifies our ability to comprehend large systems by presenting them at a level of abstraction at which a system's high-level design can be easily understood [20, 35]. Moreover, at its best, architectural description exposes the high-level constraints on system design, as well as the rationale for making specific architectural choices.

2. *Reuse:* Architectural descriptions support reuse at multiple levels. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components and also frameworks into which components can be integrated. Existing work on domain-specific software architectures, reference frameworks, and architectural design patterns has already begun to provide evidence for this [8, 31].

3. *Construction:* An architectural description provides a partial blueprint for development by indicating the major components and dependencies between them. For example, a layered view of an architecture typically documents abstraction boundaries between parts of a system's implementation, clearly identifying the major internal system interfaces, and constraining what parts of a system may rely on services provided by other parts.

4. *Evolution:* Software architecture can expose the dimensions along which a system is expected to evolve. By making explicit the "load-bearing walls" of a system, system maintainers can better understand the ramifications of changes, and thereby more accurately estimate costs of modifications. Moreover, architectural descriptions separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components, by clearly distinguishing between components and mechanisms that allow them to interact. This separation permits one to more easily change connection mechanisms to handle evolving concerns about performance interoperability, prototyping, and reuse.

5. *Analysis:* Architectural descriptions provide new opportunities for analysis, including system consistency checking [2, 25], conformance to constraints imposed by an architectural style [1], conformance to quality attributes [9], dependence analysis [42], and domain-specific analyses for architectures built in specific styles [10, 15, 26].

6. *Management:* Experience has shown that successful projects view achievement of a viable software architecture as a key milestone in an industrial software development process. Critical evaluation of an architecture typically leads to a much clearer understanding of requirements, implementation strategies, and potential risks [7].

3 YESTERDAY

In the distant past of ten years ago, architecture was largely an ad hoc affair.[1] Descriptions relied on informal box-and-line diagrams, which were rarely maintained once a system was constructed. Architectural choices were made in idiosyncratic fashion – typically by adapting some previous design, whether or not it was appropriate. Good architects – even if they were classified as such within their organiza-

[1] To be sure, there were some notable exceptions. Parnas recognized the importance of system families [33], and architectural decomposition principles based on information hiding [34]. Others, such as Dijkstra, exposed certain system structuring principles [12].

tions – learned their craft by hard experience in particular domains, and were unable to teach others what they knew. It was usually impossible to analyze an architectural description for consistency or to infer non-trivial properties about it. There was virtually no way to check that a given system implementation faithfully represented its architectural design.

However, despite their informality, architectural descriptions were central to system design. As people began to understand the critical role that architectural design plays in determining system success, they also began to recognize the need for a more disciplined approach. Early authors began to observe certain unifying principles in architectural design [36], to call out architecture as a field in need of attention [35], and to establish a working vocabulary for software architects [20]. Tool vendors began thinking about explicit support for architectural design. Language designers began to consider notations for architectural representation [30].

Within industry, two trends highlighted the importance of architecture. The first was the recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems. For example, the box-and-line-diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a "pipeline," a "blackboard-oriented design," or a "client-server system." Although these terms were rarely assigned precise definitions, they permitted designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provided significant semantic content about the kinds of properties of concern, the expected paths of evolution, the overall computational paradigm, and the relationship between this system and other similar systems.

The second trend was the concern with exploiting commonalities in specific domains to provide reusable frameworks for product families. Such exploitation is based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by "instantiating" the shared design. Familiar examples include the standard decomposition of a compiler (which permits undergraduates to construct a new compiler in a semester), standardized communication protocols (which allow vendors to interoperate by providing services at different layers of abstraction), fourth-generation languages (which exploit the common patterns of business information processing), and user interface toolkits and frameworks (which provide both a reusable framework for developing interfaces and sets of reusable components, such as menus and dialogue boxes).

4 TODAY

Much has changed in the past decade. Although there is wide variation in the state of the practice, generally speaking, architecture is much more visible as an important and explicit design activity in software development. Job titles now routinely reflect the role of software architect; companies rely on architectural design reviews as critical staging points; and architects recognize the importance of making explicit tradeoffs within the architectural design space.

In addition, the technological basis for architectural design has improved dramatically. Three of the important advancements have been the development of architecture description languages and tools, the emergence of product line engineering and architectural standards, and the codification and dissemination of architectural design expertise.

4.1 Architecture Description Languages and Tools

The informality of most box-and-line depictions of architectural designs leads to a number of problems. The meaning of the design may not be clear. Informal diagrams cannot be formally analyzed for consistency, completeness, or correctness. Architectural constraints assumed in the initial design are not enforced as a system evolves. There are few tools to help architectural designers with their tasks.

In response to these problems a number of researchers in industry and academia have proposed formal notations for representing and analyzing architectural designs. Generically referred to as "Architecture Description Languages" (ADLs), these notations usually provide both a conceptual framework and a concrete syntax for characterizing software architectures [9, 30]. They also typically provide tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.

Examples of ADLs include Adage [10], Aesop [15], C2 [28], Darwin [26], Rapide [25], SADL [32], UniCon [39], Meta-H [6], and Wright [3]. While all of these languages are concerned with architectural design, each provides certain distinctive capabilities: Adage supports the description of architectural frameworks for avionics navigation and guidance; Aesop supports the use of architectural styles; C2 supports the description of user interface systems using an event-based style; Darwin supports the analysis of distributed message-passing systems; Meta-H provides guidance for designers of real-time avionics control software; Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations; SADL provides a formal basis for architectural refinement; UniCon has a high-level compiler for architectural designs that supports a mixture of heterogeneous component and connector types; Wright supports the formal specification and analysis of interactions between architectural components.

Recently the proliferation of capabilities of ADLs has prompted an investigation of ways to integrate the notations and tools into larger ensembles. One of the results has been an architectural interchange language, called Acme, which provides a simple framework for describing architectural structure and a flexible annotation mechanism for adding semantics to that structure [18]. (Acme can be viewed as the XML of architectural description.) Acme also supports the definition of styles and enforcement of design constraints through its tools.

More ambitious still, a number of researchers have begun to look at additional ways of integrating architecture-based tools. The result promises to be a flexible "ADL Toolkit" that will allow practitioners to create domain-specific architectural design environments largely by combining the capabilities of existing tools. Other efforts have sought to integrate architectural development tools with other tool-supported aspects of software development, such as requirements elicitation and resolution [7].

Languages explicitly designed with software architecture in mind are not the only approach. There has been considerable interest in using general-purpose object design notations for architectural modeling [8, 24]. Moreover, recently there have been a number of proposals that attempt to show how the concepts found in ADLs can be mapped directly into an object-oriented notation like UML [29, 23, 17]. These alternatives are illustrated in Figure 2.

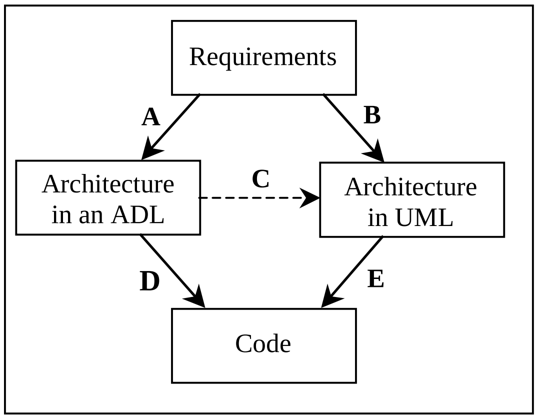

Figure 2: Software Architecture as a Bridge

Path A-D is one in which an ADL is used as the modeling language. Path B-E is one in which UML is used as the modeling notation. Path A-C-E, is one in which an architecture is first represented in an ADL, but then transformed into UML before producing an implementation.
Using a more general modeling language such as UML has the advantages of providing a notation that practitioners are more likely to be familiar with, and providing a more direct link to object-oriented implementations and development tools. But general-purpose object languages suffer from the problem that the object conceptual vocabulary may not be ideally suited for representing architectural concepts, and there are likely to be fewer opportunities for automated analysis of architectural properties.

4.2 Product Lines and Standards

As noted earlier, one of the important trends has been the desire to exploit commonality across multiple products. Two specific manifestations of that trend are improvements in our ability to create product lines within an organization and the emergence of cross-vendor integration standards.

With respect to product lines, a key challenge is that a product line approach requires different methods of development. In a single-product approach the architecture must be evaluated with respect to the requirements of that product alone. Moreover, single products can be built independently, each with a different architecture.

However, in a product line approach, one must also consider requirements for the *family* of systems, and the relationship between those requirements and the ones associated with each particular instance. Figure 3 illustrates this relationship. In particular, there must be an up-front (and on-going) investment in developing a reusable architecture that can be instantiated for each product. Other reusable assets, such as components, test suites, tools, etc., typically accompany this.

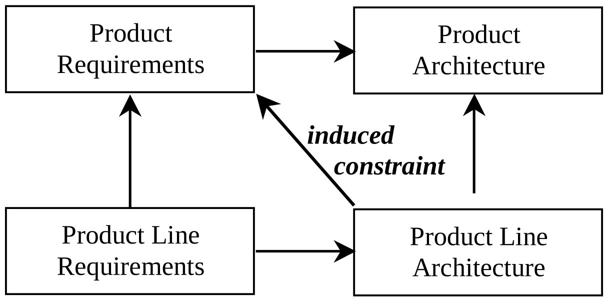

Figure 3: Product Line Architectures

Although product line engineering is not yet widespread, we are beginning to have a better understanding of the processes, economics, and artifacts required to achieve the benefits of a product line approach. A number of case studies of product line successes have been published. (For example, see [13].) Moreover, organizations such as the Software Engineering Institute are well on their way towards providing concrete guidelines and processes for the use of a product line approach [37].

Like product line approaches, cross-vendor integration standards require architectural frameworks that permit a system developer to configure a wide variety of specific systems by instantiating that framework. Integration standards typically provide the system glue (both conceptually and through run time infrastructure) that supports integration of parts provided by multiple vendors. Such standards may be formal international standards (such as those sponsored by IEEE or ISO), or ad hoc and de facto standards promoted by an industrial leader.

A good example of the former is the High Level Architecture (HLA) for Distributed Simulation [4]. This architecture permits the integration of simulations produced by many vendors. It prescribes interface standards defining services to coordinate the behavior of multiple semi-independent simulations. In addition, the standard prescribes requirements on the simulation components that indicate what capabilities they must have, and what constraints they must observe on the use of shared services.

An example of an ad hoc standard is Sun's Enterprise JavaBeans^{TM} (EJB) architecture [27]. EJB is intended to support distributed, Java-based, enterprise-level applications, such as business information management systems. Among other things, it prescribes an architecture that defines a vendor-neutral interface to information services, including transactions, persistence, and security. It thereby supports component-based implementations of business processing software that can be easily retargeted to different implementations of those underlying services.

4.3 Codification and Dissemination

One early impediment to the emergence of architectural design as an engineering discipline was the lack of a shared body of knowledge about architectures and techniques for developing good ones. Today the situation has improved, due in part to the publication of books on architectural design [5, 8, 22, 36, 40] and courses [21].

A common theme in these books and courses is the use of standard architectural *styles*. An architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary. For example, a pipe-and-filter style might specify vocabulary in which the processing components are data transformers (filters), and the interactions are via order-preserving streams (pipes). Constraints might include the prohibition of cycles. Semantic assumptions might include the fact that pipes preserve order and that filters are invoked non-deterministically.

Other common styles include blackboard architectures, client-server architectures, event-based architectures, and object-based architectures. Each style is appropriate for certain purposes, but not for others. For example, a pipe-and-filter style would likely be appropriate for a signal processing application, but not for an application in which there is a significant requirement for concurrent access to shared data [38]. Moreover, each style is typically associated with a set of associated analyses. For example, it makes sense to analyze a pipe-and-filter system for system latencies, whereas transaction rates would be a more appropriate analysis for a repository-oriented style.

The identification and documentation of such styles (as well as their more domain-specific variants) enables others to adopt previous architectural patterns as a starting point. In that respect, the architectural community has paralleled other communities in recognizing the value of established, well-documented patterns, such as those found in [14].

While recognizing the value of stylistic uniformity, realities of software construction often force one to compose systems from parts that were not architected in a uniform fashion. For example, one might combine a database from one vendor, with middleware from another, and a user interface from a third. In such cases the parts do not always work well together – in large measure because they make conflicting assumptions about the environments in which they were designed to work [16]. This has led to recognition of the need to identify architectural strategies for bridging mismatches. Although, we are far from having well understood ways of detecting such mismatch, and of repairing it when it is discovered, a number of techniques have been developed [11].

5 TOMORROW

What about the future? Although software architecture is on a much more solid footing than a decade ago, it is not yet established as a discipline that is taught and practiced universally across the software industry. One reason for this is simply that it takes time for new approaches and perceptions to propagate. Another reason is that the technological basis for architecture design (as outlined earlier) is still immature. In both of these areas we can expect that a natural evolution of the field will lead to steady advances.

However, the world of software development and the contexts in which software is being used are changing in significant ways. These changes promise to have a major impact on how architecture is practiced. In the remainder of this section I consider three of the more prominent trends and their implications for the field of software architecture.

5.1 Changing Build-Versus Buy Balance

Throughout the history of software engineering a critical issue in the development of systems has been the decision about what parts of the system to obtain elsewhere and what parts to build in-house. Externally acquired parts have the advantage of saving development time. But they also have the disadvantages of often incompletely satisfying the need, and of being less under the control of the developing organization.

While the issue itself has not changed, economic pressures to reduce time-to-market are drastically changing the balance. For an increasing number of products, introducing a product a month early may be the difference between success and failure. In such situations building systems using software that others have written becomes the only feasible choice. It is not unusual these days for software developers to use a thousand (or even ten thousand) lines of externally developed code for every line they write. Indeed, many companies are rapidly finding themselves more in the position of system integrators than software developers. That is, most of the code they *do* write is "glue" code that causes a set of system components to work together.

For many companies the situation is exacerbated by economic trends toward mergers and acquisitions as the primary avenue of growth. Rather than buying individual software parts or products, companies simply absorb other companies in total. Integration now becomes an even more serious and difficult problem.

There are a number of consequences for software architecture. First, this trend heightens the need for industry-wide standards. The more commonality, the more likely third party software systems will work together. This trend is evident in the rising popularity of "component-based" software development [43]. By picking components that agree on a common architectural framework, such as COM, JavaBeans, or CORBA, many of the problems of architectural mismatch are alleviated.

However, "component-based" engineering is only part of the answer. Today's component technologies work at a fairly low level of architectural abstraction – essentially at the level of procedure call between objects. To obtain more significant integration will require higher-level architectural standards. This is likely to lead us from "component-based" engineering to "architecture-based" engineering, a shift that will emphasize the role of domain-specific integration architectures (such as EJB or HLA, mentioned earlier) in promoting composability. Thus, the trend toward architectural standards, noted earlier, is likely to become even more pronounced.

Second, this trend is leading to new software subcontracting processes. When integration is the critical issue, we can expect that externally contracted software will be held to much higher standards of architectural conformance. In many cases the standards will be commercial or governmental standards. In others, it is likely that sub-contractors will need to guarantee compatibility with product line architectures. For example, the US Department of Defense requires that simulation subcontractors furnish products that are "HLA-compliant."

Third, this trend is leading toward standardization of notations and tools across vendors. When down-stream integrators are recombining systems, it is no longer acceptable to have in-house, idiosyncratic architecture descriptions and tools. This has led many to adopt languages like UML and XML for architectural modeling (cf., Section 4.1).

5.2 Network-Centric Computing

There is a clear trend from a PC-centric computational model to a network-centric model. Traditional computer use centers on PCs, using local applications and data that are installed and updated manually. Increasingly, the PC and a variety of other interfaces (handheld devices, laptops, phones) are used primarily as a user interface that provides access to remote data and computation. This trend is not surprising since a network-centric model offers the potential for significant advantages. It provides a much broader base of services than is available on an individual PC. It permits access to a rich set of computing and information retrieval services through portable computers that can be used almost anywhere (in the office, home, car, and factory). It promotes user mobility through ubiquitous access to information and services.

This trend has a number of consequences for software engineering, in general, and software architecture, in particular. Historically, software systems have been developed as closed systems, developed and operated under control of individual institutions. The constituent components may be acquired from external sources, but when incorporated in a system they come under control of the system designer. Architectures for such systems are either completely static – allowing no run time restructuring – or permit only a limited form of run time variation.

However, within the world of pervasive services available over networks, systems may not have such centralized control. The Internet is an example of such an open system: It is minimally standardized, chiefly at the level of the protocols, addresses, and representations that allow individual sites to interact. It admits of considerable variation both in the hardware that lies below these standards and the applications that lie above. There is no central authority for control or validation. Individual sites are independently administered. Individual developers can provide, modify, and remove resources at will.

For such systems a new set of software architecture challenges emerges [41]. First, is the need for architectures that scale up to the size and variability of the Internet. While many of the same architectural paradigms will likely apply, the details of their implementation and specification will need to change.

For example, one attractive form of composition is implicit invocation – sometimes termed "publish-subscribe." Within this architectural style components are largely autonomous, interacting with other components by broadcasting messages that may be "listened to" by other components. Most systems that use this style, however, make many assumptions about properties of its use. For example, one typically assumes that event delivery is reliable, that centralized routing of messages will be sufficient, and that it makes sense to define a common vocabulary of events that are understood by all of the components. In an Internet-based setting all of these assumptions are questionable.

Second, is the need to support computing with dynamically-formed, task-specific, coalitions of distributed autonomous resources. The Internet hosts a wide variety of resources: primary information, communication mechanisms, applications that can be invoked, control that coordinates the use of resources, and services such as secondary (processed) information, simulation, editorial selection, or evaluation. These resources are independently developed and independently supported; they may even be transient. They can be composed to carry out specific tasks set by a user; in many cases the resources need not be specifically aware of the way they are being used, or even whether they are being used. Such coalitions lack direct control over the incorporated resources. Selection and composition of resources is likely to be done afresh for each task, as resources appear, change, and disappear. Unfortunately, it is hard to automate the selection and composition activity because of poor information about the character of services and hence with establishing correctness.

Composition of components in this setting is difficult because it is hard to determine what assumptions each component makes about its operating context, let alone whether a set of components will interoperate well (or at all) and whether their combined functionality is what you need. Moreover, many useful resources exist but cannot be smoothly integrated because they make incompatible assumptions about component interaction. For example, it is hard to integrate a component packaged to interact via remote procedure calls with a component packaged to interact via shared data in a proprietary representation.

These problems will provide new challenges for architecture description and analysis. In particular, the need to handle dynamically-evolving collections of components obtained from a variety of sources will require new techniques for managing architectural models at run time, and for evaluating the properties of those ensembles.

Third, is the related need to find architectures that flexibly accommodate commercial application service providers. In the future, applications will be composed out of a mix of local and remote computing capabilities on each desktop, requiring architectural support that supports billing, security, etc.

Fourth, is the need to develop architectures that permit end users to do their own system composition. With the rapid growth of the Internet, an increasing number of users are in a position to assemble and tailor services. Such users may have minimal technical expertise, and yet will still want strong guarantees that the parts will work together in the ways they expect.

5.3 Pervasive Computing

A third related trend is toward pervasive computing in which the computing universe is populated by a rich variety of heterogeneous computing devices: toasters, home heating systems, entertainment systems, smart cars, etc. This trend is likely to lead to an explosion in the number of devices in our local environments – from dozens of devices to hundreds or thousands of devices. Moreover these devices are likely to be quite heterogeneous, requiring special considerations in terms of their physical resources and computing power.

There are a number of consequent challenges for software architectures. First, we will need architectures that are suited to systems in which resource usage is a critical issue. For example, it may be desirable to have an architecture that allows us to modify the fidelity of computation based on the power reserves at its disposal.

Second, architectures for these systems will have to be more flexible than they are today. In particular, devices are likely to come and go in an unpredictable fashion. Handling reconfiguration dynamically, while guaranteeing uninterrupted processing, is a hard problem.

Third, is the need for architectures that will better handle user mobility. Currently, our computing environments are configured manually and managed explicitly by the user. While this may be appropriate for environments in which we have only a few, relatively static, computing devices (such as a couple of PCs and laptops), it does not scale to environments with hundreds of devices. We must therefore find architectures that provide much more automated control over the management of computational services, and that permit users to move easily from one environment to another.

6 CONCLUSION

The field of software architecture is one that has experienced considerable growth over the past decade, and it promises to continue that growth for the foreseeable future. As architectural design matures into an engineering discipline that is universally recognized and practiced, there are a number of significant challenges that will need to be addressed. Many of the solutions to these challenges are likely to arise as a natural consequence of dissemination and maturation of the architectural practices and technology that we know about today. Other challenges arise because of the shifting landscape of computing and the needs for software: these will require significant new innovations. In this paper we have attempted to provide a high-level overview of this terrain – illustrating where we have come over the past few years, and speculating about where we need to go to meet the demands of the future.

ACKNOWLEDGEMENTS

This article would only have been possible by virtue of many long-standing interactions and collaborations with numerous colleagues and students over the past decade. Among these I would particularly like to credit past and present members of the ABLE Research Group, Bob Balzer, Barry Boehm, Drew Kompanek, Dewayne Perry, John Salasin, Mary Shaw, Dave Wile, and Alex Wolf. Software architecture would not have come nearly as far as it has were it not for generous funding from DARPA, NSF, and many industrial sponsors. My own research has been sponsored by DARPA under contracts F30602-97-2-0031, F30602-96-1-0299, F30602-96-2-0224, and F33615-93-1-1330; by NSF under contracts CCR-9109469, CCR-9633532, and CCR-9357792; and by industrial support from HP, Siemens Corporation, and Kodak Corporation.

Mary Shaw provided some of the ideas and text for the section on pervasive computing architectures. The section describing the roles of software architecture was adapted from an introductory article on software architecture co-authored with Dewayne Perry [19].

REFERENCES

1. G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering.* ACM Press, December 1993.

2. R. Allen and D. Garlan. Formalizing architectural connection. In *Proceeding of the 16th International Conference on Software Engineering*, pages 71-80. Sorrento, Italy, May 1994.

3. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.

4. S. Bachinsky, L. Mellon, G. Tarbox, and R. Fujimoto. RTI 2.0 architecture. In *Proceedings of the 1998 Spring Simulation Interoperability Workshop*, 1998.

5. L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice.* Addison Wesley, 1099, ISBN 0-201-19930-0.

6. P. Binns and S. Vestal. Formal real-time architecture specification and analysis. *10th IEEE Workshop on Real-Time Operating Systems and Software*, May 1993.

7. B. Boehm, P. Bose, E. Horowitz and M. J. Lee. Software requirements negotiation and renegotiation aids: A theory-W based spiral approach. In *Proc of the 17th International Conference on Software Engineering*, 1994.

8. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

9. P. Clements, L. Bass, R. Kazman and G. Abowd. Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality*, Austin, Texas, Oct, 1995.

10. L. Coglianese and R. Szymanski, DSSA-ADAGE: An Environment for Architecture-based Avionics Development. In *Proceedings of AGARD'93*, May 1993.

11. R. Deline. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon University, December 1999.

12. E. W. Dijkstra. The structure of the "THE" – multiprogramming system. *Communications of the ACM,* 11(5):341-346, 1968.

13. P. Donohoe, editor. *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer Academic Publishers, 1999.

14. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

15. D. Garlan, R. Allen and J. Ockerbloom. Exploiting style in architectural design environments. In *Proc of SIGSOFT'94: The second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 170-185. ACM Press, December 1994.

16. D. Garlan, R. Allen and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17-28, November 1995.

17. D. Garlan, A. J. Kompanek and P. Pinto. Reconciling the needs of architectural description with object-modeling notations. Technical report, Carnegie Mellon University, December 1999.

18. D. Garlan, R. T. Monroe and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169-183, Ontario, Canada, November 1997.

19. D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

20. D. Garlan and M. Shaw. An Introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1-39, Singapore, 1993. World Scientific Publishing Company.

21. 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.

22. C. Hofmeister, R. Nord and D. Soni. *Applied Software Architecture*. Addison Wesley, 2000.

23. C. Hofmeister, R. L. Nord and D. Soni. Describing software architecture with UML. *In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

24. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pages 42-50, November 1995.

25. D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Veera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4): 336-355, April 1995.

26. J. Magee, N. Dulay, S. Eisenbach and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference*, ESEC'95, September 1995.

27. V. Matena and M. Hapner. Enterprise JavaBeans™. Sun Microsystems Inc., Palo Alto, California, 1998.

28. N. Medvidovic, P. Oreizy, J. E. Robbins and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *SIGSOFT'96: Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*. ACM Press. Oct 1996.

29. N. Medvidovic and D. S. Rosenblum. Assessing the suitability of a standard design method for modeling software architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999.

30. N. Medvidovic and R. N. Taylor. Architecture description languages. In *Software Engineering ESEC/FSE'97*, Lecture Notes in Computer Science, Vol. 1301, Zurich, Switzerland, Sept 1997. Springer.

31. E. Mettala and M. H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9. Carnegie Mellon Univ., Jun 1992.

32. 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.

33. D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5:128-138, March 1979.

34. D. L. Parnas, P. C. Clements and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*. SE-11(3):259-266, March 1985.

35. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, October 1992.

36. E. Rechtin. *Systems architecting: Creating and Building Complex Systems*. Prentice Hall, 1991.

37. CMU Software Engineering Institute Product Line Program. http://www.sei.cmu.edu/activities/plp/, 1999.

38. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of COMPSAC 1997*, August 1997.

39. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnick. Abstractions for software architecture and tools to support them. *IEEE Trans on Software Engineering.* 21(4):314-335. April 1995.

40. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

41. Mary Shaw. Architectural Requirements for Computing with Coalitions of Resources. 1st Working IFIP Conf. on Software Architecture, Feb 1999 http://www.cs.cmu.edu/~Vit/paper_abstracts/Shaw-Coalitions.html.

42. J. A. Stafford, D. J. Richardson, A. L. Wolf. Aladdin: A Tool for Architecture-Level Dependence Analysis of Software. University of Colorado at Boulder, Technical Report CU-CS-858-98, April, 1998.

43. C. Szyperski. Component Software: *Beyond Object-Oriented Programming*. Addison-Wesley, 1998.