Technical Report
CMU/SEI-96-TR-008
ESC-TR-96-008

Coming Attractions in Software Architecture

Paul C. Clements

January 1996

Technical Report
CMU/SEI-96-TR-008
ESC-TR-96-008
January 1996

Coming Attractions in Software Architecture

Paul C. Clements

Software Architecture Technology Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(Signature on File)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

1. Introduction 1

2 Architecture-Based Development 3

3 Most Promising Work 7
3.1 Architecture Design or Selection 7
3.2 Architecture Representation 10
3.3 Architecture Evaluation and Analysis 11
3.4 Architecture-Based Development and Evolution 12
3.5 Architecture Recovery 12

4 What Do Other Experts Say? 15
4.1 Garlan and Perry 15
4.2 Nierstrasz and Meijler 15
4.3 1995 Monterey Workshop 15

5 Time Predictions 17

6 Acknowledgments 19

References 21

Coming Attractions in Software Architecture

Abstract: Software architecture is a field of study enjoying unprecedented growth and interest. This report identifies a set of promising lines of research related to software architecture and architecture-based system development that are expected to lead to advances available soon to practitioners. Some of the goals of software architecture are enumerated, and the investigatory efforts are structured according to work in the design or selection and creation, representation, evaluation and analysis, utilization, or legacy recovery of software architectures. Promising research is described in each area. These opinions are correlated with those of other experts in the field. Finally, a timeline for achieving some of the predicted results is offered.

1 Introduction

This technical report identifies a set of promising lines of research in the field of architecture-based software development that are expected to lead to advances that will be available to practitioners over the next five to ten years. Most members of the set are based upon current (but embryonic) efforts in the field; a few are based on the judgement of the author. Promising areas are defined to be those that seem to hold potential for most positively affecting the development and evolution processes for large-system software over the next ten years or so, and which are now the subject of only limited or localized work. It was felt that focusing on under-explored areas would be of more interest to the community than treating widespread efforts. Thus, if an area of research is not mentioned, the implication is that the current effort in that area is proceeding apace, not that it is a poor bet. An example of the latter is architecture description language (ADL) development, currently being prosecuted to good effect by over a dozen research efforts.

Since this report attempts to summarize a changing field, it is intended to be living; it will be updated at regular intervals. Comments are invited and suggestions for inclusions are welcome and solicited. Please contact the author via electronic mail at

swarch@sei.cmu.edu

Before describing promising research in any field, it is necessary to submit a world view or vision toward which the field is or should be progressing, and against which work in progress may judged. Section 2 provides a glimpse of such a world view, proposing that the promise of software architecture may be viewed as facilitating the following capabilities, each of which can lead to significant improvement in the development and deployment of large-scale system software:

- component-based development
- early quality prediction
- product line development
- separation of functionality from interconnection
- constraining the design space

Section 3 discusses some of the most promising work towards achieving that world view. Section 4 describes the current research agenda as seen by other experts in the field. Section 5 proposes a time frame to each development suggested in Section 3.

2 Architecture-Based Development

Software architecture is, roughly, a view of a system that includes the system's major components, the behavior of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the system's mission. The architectural view is an abstract view, bringing with it the higher level of understanding, and suppression and deferral of detail inherent in most abstractions.

The study of software architecture, although recently enjoying significant impetus, is in large part rooted in a study of software structure that began in 1968 with Edsger Dijkstra's landmark operating system paper. Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result [Dijkstra 68]. David Parnas pressed this line of observation with his contributions concerning information-hiding modules, software structures, and program families, all of which stressed qualities of software measurable in terms of economies to the development and maintenance processes [Parnas 72, Parnas 74, Parnas 76].

All of the work in the field of software architecture can be seen as evolving toward a paradigm of software development based on principles of large-scale, component-based system construction, and for exactly the same reasons given by Dijkstra and Parnas: structure matters. Choosing an appropriate structure, with appropriate coordination mechanisms among the structural parts, yields economies of production without sacrificing required performance or correctness attributes. This paradigm has not yet crystallized into a codified form, but the work seems to reflect a systematic belief in the following tenets, some of which remain speculative:

- **Systems can be built in a rapid, cost-effective manner by importing (or generating) large externally developed components.** Former software paradigms have focused on *programming* as the prime activity, with progress measured in lines of code. Architecture-based development focuses on *assembling components* that are likely to have been developed separately, even independently, from each other. Integration becomes the critical activity.

  Areas of current investigation addressing this tenet include large-scale software reuse, component-based development, COTS system development, COTS system integration, interface standards and specification work, parameterized programming, and infrastructural frameworks into which components can be inserted.

- **It is possible to predict certain qualities about a system by studying its architecture, even in the absence of detailed design and code.** Performance is largely a function of the frequency and nature of inter-component communication, in addition to performance characteristics of the components themselves, and hence can be predicted by studying the architecture of a system[1]. A non-runtime quality attribute such as maintainability is largely a function of the locality of anticipated changes, which can be catalogued in terms of which architectural components such changes would affect.

Work in architectural analysis and modelling techniques exemplifies this tenet [Kazman 94].

- **Entire product lines can be developed by sharing a common architecture. Large-scale reuse is possible through architectural-level planning.** Product lines are groups of related systems that, together, fill a market niche. They are derived from what Parnas referred to in 1976 as program families [Parnas 76]. Parnas wrote that a fielded system is a leaf in a decision tree, where each node represents a design decision. Deriving a second instance of the system involves, at best, backing up to the lowest common decision point and re-traversing to reach the new leaf. At worst, it means starting over. Therefore, it pays to carefully order the design decisions one makes so that the most likely to be changed occur latest in the process. In an architecture-based development of a product line, one chooses an architecture (or a family of closely related architectures), and a set of generic components that will serve all or nearly all envisioned members. These choices represent decisions near the top of Parnas's decision tree. Variations among members are handled by late binding of parameters, swapping in interchangeable components, etc.[2]

  The work in domain analysis, domain engineering, component-based design methodologies, and reuse all support this tenet of the paradigm. Work defining disciplined, architecture-based evolution strategies also comes into play because it regards the system before and after a change as two separate, but closely related, members of the same program family.

- **The functionality of a component can be separated from its component interconnection mechanisms for good reasons.** Traditional design approaches have been primarily concerned with the functionality of components. Architecture work seeks to add a second concern: how a component interacts, coordinates, cooperates, and communicates with other components. The stated goal is to recognize the different fundamental qualities imparted to systems by these various interconnection strategies and to encourage informed choices. However, the result is a separation of concerns, which introduces the possibility of building architectural infrastructure to automatically implement the architect's eventual choice of mechanism. The binding of this decision may be delayed and/or easily changed[3]. Thus, prototyping and large-scale system evolution are both supported. Although proponents of this view speak of "first-class connectors," they are actually making it possible for the question of connectors to be ignored, or at

---

1. Some may ask if software performance is not a function of the speed of the underlying hardware and thus unpredictable with only a software architecture. The answer is not always. Sometimes performance is dictated by external requirements. For example, a display may be required to be updated every 30 milliseconds because that is the rate at which humans perceive continuous motion from discrete frames. This in turn will determine the performance requirements for the software components that prepare the data and drive the display. These performance constraints may be only loosely tied to capabilities of specific hardware; the software components perform in step with a real-time clock, rather than as fast as the hardware allows. Only a loose assumption about the hardware—namely, that the hardware is sufficiently speedy to allow the components their necessary real-time performance—is required. The looser the ties to the hardware, the more fidelity architecture-level performance predictions will have. On the other hand, in systems where performance is tightly constrained, it is more likely that hardware decisions will be made early on, in which case that information can be used to aid in early performance predictions.

2. Note that application generators circumvent this tenet to some extent. With an application generator, early design decisions are no longer the hardest to change, since the automation provided by the generator obviates the cost of the change. Also, Section 3.2 discusses an architecture-based development environment that allows rapid revision of architecture-level decisions.

least deferred, in many cases [Shaw 94]. This contrasts to the programming paradigm, where connection mechanisms are chosen very early in the design cycle, without much thought, and are nearly impossible to change. Areas addressing this aspect include architecture description languages that embody connection *abstractions*, as opposed to mechanisms.

Supporting work in this area includes design of architecture description languages, specification of component interfaces (especially thread-of-control aspects), formal models of composition and interconnection, and those languages and/or environments that feature automatic "glue code" generation. For an example of a formal model of composition, see "Correctness and Composition of Software Architectures," [Moriconi 94].

- **Less is more: it pays to restrict the vocabulary of design alternatives.** David Garlan and Mary Shaw's work in cataloguing architectural styles teaches us that, although computer programs may be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small set of choices when it comes to program cooperation and interaction [Garlan 93]. Advantages include enhanced reuse, more capable analysis, shorter selection time, and greater interoperability.

  ADLs and case study work both support this aspect by helping to identify useful members of a restricted vocabulary. Architecture analysis and evaluation techniques also apply because they help developers choose among alternatives. The blossoming design pattern community is a lower level offshoot of this tenet, giving us ways to describe and represent patterns of interaction among a set of components. Finally, work in the theory of component interfaces helps identify the information channels across which components interact.

[3] Mechanisms may include subroutine invocation with parameters, subroutine invocation with global data, implicit invocation via event-signalling, implicit invocation via blackboard, any flavor of process synchronization, and others. For examples, see "Formalizing Architectural Connection," and *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis* [Allen 94, Fernandez 93].

3 Most Promising Work

Problem areas in architecture tend to be clustered around the following five themes arranged in terms of designing, building, and maintaining or evolving a system based on its architecture:

- **Architecture design or selection:** How to create or select an architecture based on a set of functional, performance, and quality requirements.
- **Architecture representation:** How to communicate an architecture. This problem has manifested itself as one of representing architectures with linguistic facilities, but the problem also includes selecting the set of information to be communicated; that is, represented with a language.
- **Architecture evaluation and analysis:** How to analyze an architecture to predict qualities about systems that manifest it. A similar problem is how to compare and choose between competing architectures.
- **Architecture-based development and evolution:** How to build and maintain a system given a representation of what is confidently believed to be a sound architecture that will solve the problem at hand. The components may or may not already exist; if so, they may or may not be initially compatible with each other.
- **Architecture recovery:** How to evolve a legacy system when changes may affects its architecture; for systems lacking trustworthy architectural documentation, this will first involve “architectural archaeology” to extract its architecture.

We will discuss each area in turn.

3.1 Architecture Design or Selection

Technologies to support the creation or selection of an architecture for a system can be seen to exist along a spectrum, shown in Figure 1. At one end are ad hoc techniques, in which experienced and/or talented designers conjure up an architecture in a largely unrepeatable fashion. Farther up the spectrum lie reuse techniques, from previously used architectures, to architectures populated with reusable components, to architectures populated with tailorable and parameterized components. Architectures based on frameworks such as MacApp (a development environment for MacIntosh application programs) or the CORBA object management architecture (OMA) lie in this region [Object Management Group 95]. They offer differing levels of “plug-in-and-run” completeness and application independence, but both provide their own architectural reference models and support those models with executable software components. At the high end of the spectrum lie partial and pure application generators. An application generator is a program that incorporates knowledge about the relevant application domain and, given as its input a set of requirements for a particular mem-

ber of that domain, generates software that implements that domain member. A pure application generator produces a turnkey system and renders moot the question of architecture to the user of the generator. Less encompassing generators produce components that must be integrated into the eventual system; the Unix-based parser generator YACC is an example.

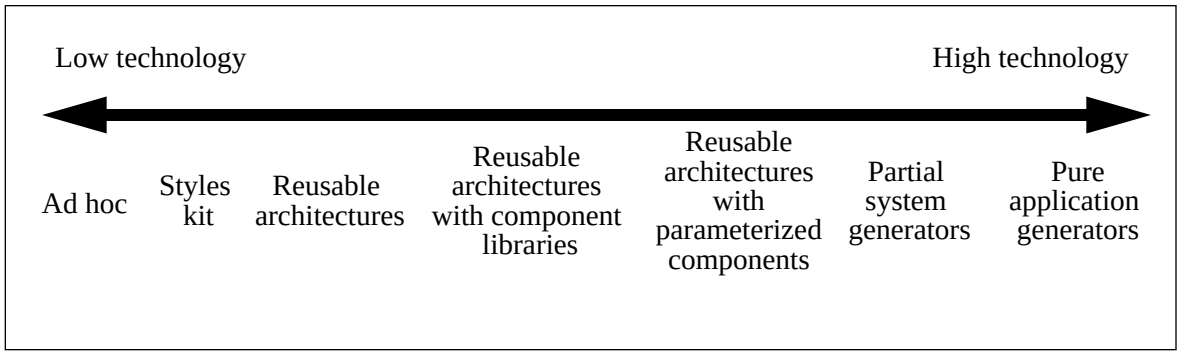

Figure 1: Technology spectrum for architecture selection and creation

Promising work in this area consists of case studies and application generator technology.

Case studies: For approaches that are not ad hoc to be successful, experience must be gained in building reusable architectures in a particular "architecture domain" [4]. Central to this work is understanding the relationship between requirements and architecture; in particular, understanding how a system is "driven" into a particular architecture domain. What role do quality requirements play? Performance requirements? Organizational history or constraints? Case studies can provide insight into how requirements and context interact with each other in order to produce an architecture.

As case studies are promulgated, the following results can be expected:

- Agreement will emerge as to a taxonomy of systems' problem spaces. A gross taxonomy can be said to exist today. Is a system hard-real time or not? Is it required to be distributed or not? These and other coarse-grained discriminators that currently exist may be seen to fundamentally affect the type of system fielded, and will evolve to more sophisticated and fine-grained characterizations of the problem space in the future. Jackson's work on problem frames is a start in this direction [Jackson 94a]. Application-specific problem

4. Unlike an application domain, which refers to a customer- or user-oriented set of products, such as avionics programs, an architecture domain refers to a set of programs implementable via the same architectural pattern, interconnection mechanisms, etc. Some avionics programs may be implemented with a single-processor timing-loop structure; others may be highly parallel with message-passing interaction. These examples represent two different architecture domains, each of which may be populated by other systems from different application domains.

taxonomies are also emerging, such as “A Taxonomy of Computer Program Security Flaws,” [Landwehr 94].

- Agreement will emerge as to a taxonomy of systems’ context space. What are the organizational influences on architecture? What effect does the prior experience of senior designers have? As these and other influences emerge and are systematically captured, business case strategies can be built based on an organization’s technical background, infrastructure, and capability.
- Agreement will emerge as to a taxonomy of systems’ solution spaces. The work in architectural styles and solution viewpoints represents early promising work, as do taxonomies of coordination mechanisms [Perry 92, Garlan 93, Shaw 95, Fernandez 93]. Work in finding, capturing, formalizing, and exploiting design patterns, and identifying supporting technology for pattern-based development, also represents an important approach that is young but growing in importance. For examples, see “Patterns Generate Architectures,” and *Design Patterns, Elements of Object-Oriented Software* [Beck 94, Gamma 95].

In theory, case study work could converge into production of design guidebooks, like those found in other engineering disciplines and which seem to be emerging in the design patterns community. The goal is to produce systematic, reliable design guidance: assistance in asking appropriate architecture-determining questions about requirements and being directed to architectures or architectural decisions that plausibly solve the problem. For example, see *Software Architectures for Shared Information Systems* [Shaw 93].

Application generator technology: In order to produce a pure application generator for a domain, an alphabet of primitive components must be built to be combinable in flexible, arbitrary ways. Component identification, component composition, and mapping to a given physical architecture are the driving problems. Currently, a leader in application generator technology is represented by the GenVoca method [Batory 94, Beck 94]. GenVoca has an innovative approach to component composition and has been successfully applied to various application domains. Component identification is currently ad hoc. The codification of this part of the process, by marrying domain analysis methods with architecture component identification, should produce a dramatic improvement in our ability to build generators for domains for which such a possibility was only recently unthinkable. For example, the existence of parser generators, optimization generators, program flow graph generators, and the like renders it unthinkable to build a compiler from scratch today. Similarly, it may soon be the case that nobody will ever build from scratch an avionics programs, a database management system, a military command center, or a software engineering environment because of the existence of generators to produce application-standard components attached to an application-standard architectural framework. Promising generator work includes the construction of generator generators, generators with user-level interfaces

(including graphical specification languages), and generators that allow a declarative specification of the target computing environment.

3.2 Architecture Representation

A system's architecture serves many stakeholders and it must be communicated to each of them. For example, no matter how components are chosen, that architectural choice becomes immortalized in the developing organization's work breakdown structure, team assignments and structure, unit test plans, integration test plans, project schedule, and maintenance and evolution plans. The architecture provides the medium for inter-team cooperation and communication. It serves as the basis for early modelling, evaluation, and prediction of performance, schedulability, feasibility, and resource allocation.

Communicating an architecture to a stakeholder becomes a matter of representing it in an unambiguous, readable form that contains the information appropriate to that stakeholder. While development of architecture languages is proceeding apace—there are at least two dozen languages capable of, if not developed explicitly for, representing architectural information—there is less attention being paid to the following areas:

Infrastructures to support ADL development. Most ADLs share a set of common concepts. Building tools to support an ADL involves solving a common set of problems. Development of an ADL development environment would facilitate the rapid production of ADLs and supporting tools, thus allowing good ideas to come to market faster. Garlan's Aesop/ACME work represents an important contribution to this area, as do efforts to formalize what we mean by architecture so that representation and representation-based analysis capabilities in languages can be enhanced [Garlan 94, Inverardi 95].

Integration of ADL information with other life-cycle products. As ADLs mature, they will take a more prominent role in the litany of life-cycle products (such as detailed design documents, test cases, etc.). Encouragement should be given to early consideration of the relationship that an architecture description (and the ADL tool to render it) will bear to these other documents (and the tools that produce/maintain them). For example, what test cases might be generated for a system based on a description of its components and interconnection mechanisms? What kind of and how much executable code can be automatically generated? How can traceability of architecture to requirements be established? How can architectural patterns, like design patterns, be rapidly imported into the architecture [Gamma 95]? This work could culminate in the complete integration of architecture descriptions into the development environment, giving rise to a sort of "architectorium." This can be thought of as an exploration envi-

ronment in which architectures are drafted, validated via mapping to requirements, their implications explored via analysis or rapid prototyping, alternatives suggested in an expert-system-like fashion, and project infrastructures necessary for development (e.g., work schedule templates, component-based configuration control libraries, test plans, etc.) are generated.

3.3 Architecture Evaluation and Analysis

One of the promises of architecture as a field of study is that it is possible to predict qualities of a finished system just by studying its architecture. If this is true, then it will ameliorate the syndrome whereby validating early design decisions occurs only when it's too late to change them. Two aspects of this problem are ripe for breakthrough work.

Quantification of functional and afunctional qualities. An architecture is chosen because it achieves functional properties (behavioral, performance, security, etc.) and afunctional properties (the ability to support maintenance and evolution, product line building, and low time-to-market development) that are important to the developer. In order to evaluate an architecture against attributes of significance, it must be possible to express those attributes in a quantitative way. Current evaluation methods such as SAAM finesse the issue through the use of scenarios; quality attributes are never expressed directly at all [Kazman 94]. To see if an architecture is maintainable, an architect poses a set of specific change scenarios and evaluates the architecture against each of those. (Performance benchmarks are the runtime analogy to scenarios, pointing out the absence of believable generic performance metrics.)

Practical verification strategies. A number of environments exist in which an architectural rendition can be used to generate a simulation of the system. For example, see "Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems," [Luckham 93]. However, simulation is inherently a weak validation tool in that it only presents a single execution of the system at a time; like testing, it can only show the presence rather than absence of faults. More powerful are verifiers or theorem provers that are able to compare a desired safety assertion against all possible executions of a program at once. Current generation verifiers, such as the one that supports Modechart, are limited in power because they suffer from state-explosion problems, rendering them useful only for small problems or subsets of actual systems [Jahanian 86, Jahanian 94]. Inroads are being made, but progress must continue [Burch 90, Jackson 94]. Theorem provers are labor intensive and are limited in scope. Work to make proof of correctness practical for large systems is vital. Given an architecture (in the form of components, connections, functional and performance information about the components, and built-in semantic knowledge about connector types) a

verifier could assure developers that performance requirements, deadline satisfaction, resource utilization constraints, and security and invariant safety conditions were all achievable, or point out places in the architecture where they were not.

3.4 Architecture-Based Development and Evolution

Given a satisfactory architecture, a system must still be built that reflects that architecture with complete integrity and fidelity. Besides the “architectorium” environment mentioned earlier, one other area of investigation offers special promise.

New architecture-based design methods. Architecture-based design represents a development paradigm that differs in fundamental ways from current alternatives; in many ways, it is as different as object-oriented development (OOD) was from its predecessors. OOD plays host to a rich community of methodologists and practitioners, trading information about application and practice of object-oriented technology. Architecture-based technology will need to nurture a similarly fertile “thought environment” by establishing a culture in which architectural issues and ideas flourish. Workshops with architectural themes are a start. For example, see “First International Workshop on Architectures for Software Systems” [Garlan 95a]. A technical issue that will need to be addressed early is crafting a precise articulation of (possibly more than one) architecture-based design paradigm and working out the associated process issues. An example of a new paradigm is represented by the adaptive programming approach of Demeter [Lieberherr 96].

Component interfaces. Currently, the interface to a component is largely a collection of syntactic information (names of method programs, type and number of parameters, etc.) with scant semantic information (global data affected, exceptions possibly raised, and some informally expressed description of what the component does). This information is wholly inadequate to effectively use a component that was developed outside the scope of the using project. Performance information, security information, reliability information, assumptions about threads of control, assumptions about supporting facilities present elsewhere in the system at compile-time or link-time or run time, and other critical information is required and practically never provided. Work is needed to categorize interface information, aim it to specific audiences that each have different needs, understand when each type is needed, and explore how to best express it.

3.5 Architecture Recovery

Given a legacy system without an architecture description, how can changes be made that will not corrupt the design? What if a needed change is so severe that it cannot

be performed within the framework of the given architecture? Work is needed to mature the following areas:

Architecture archaeology. Reliable technology is needed to identify components of different types, for example, processes, modules, objects, and the ways in which they interact with each other. Object finders are an example of this. Program dependency graph generators (used extensively in optimizing compilers) are another example that might be modified to help uncover an architecture. In principle, tools like this often encounter computational complexity problems that will prevent total solutions. However, work can certainly be done to solve special cases of the problem and to understand what kind of programmer-provided annotations will help make the problem tractable. For example, a program's call- or data-flow graph typically looks radically different (and much more organized) if the exception-handling flows are removed. Annotations could help identify such cases. Another promising approach is the middle-of-the-road line taken by Murphy, Notkin, and Sullivan, in which automation does not recover an architecture but checks a human user's assertion (or guess) about the architecture [Murphy 94].

Architecture migration technology. Work is needed to understand, given an architecture, what changes it will support and what changes are outside its scope. For out-of-scope changes, technology is needed that will provide disciplined, orderly ways to evolve an architecture. Fluid architectures are an example of this: if the responsibilities of one component could be migrated to another in an orderly fashion, perhaps such a migration could accommodate the change, salvage the architecture obviating the need to start over, and still result in an architecture that resembled the earlier one and was orderly instead of ad hoc [Scherlis 94, Scherlis 95]. Also needed is an understanding about "closeness of fit," to handle the case where components are almost, but not quite, compatible with each other. Finally, standard engineering practices must be found that migrate the out-of-line components into the dominant architectural framework.

4 What Do Other Experts Say?

4.1 Garlan and Perry

David Garlan and Dewayne Perry wrote a guest introduction for the April 1995 issue of *IEEE Transactions on Software Engineering* that was devoted to software architecture that outlines the most promising research areas [Garlan 95b]. David Garlan reissued the list in the June 1995 *ACM Computing Surveys* [Garlan 95c]. The list consists of the following:

- architectural description languages
- formal underpinnings of software architecture (mathematical foundations, formal characterization of extra-functional properties such as maintainability, theories of interconnection, etc.)
- architectural analysis techniques
- architectural development methods
- architecture recovery and reuse
- architectural codification and guidance
- tools and environments for architectural design
- case studies

4.2 Nierstrasz and Meijler

Nierstrasz and Meijler identify three important areas of work in component-based development [Nierstrasz 95]. These areas are rather coarse-grained, but nevertheless correlate well with other views:

- composition models (e.g., for describing component frameworks or interaction mechanisms)
- composition languages
- tools and methods

4.3 1995 Monterey Workshop

Finally, the 1995 Monterey Workshop on Formal Methods and Software Architecture, held at the Naval Postgraduate School, concluded that directions for future research included “investigating methods for effectively representing design rationale so that it

can be used to provide automated decision support. Some issues mentioned were how to capture design knowledge and how to model design decisions." [Berzins 95]

5 Time Predictions

This section, by way of Figure 2, predicts progress in each of the discussed areas over a five- to ten-year period. Its purpose is mostly to stimulate discussion about likely paths to achieve the goals. It should not be viewed as a claim to clairvoyance on the part of the author.

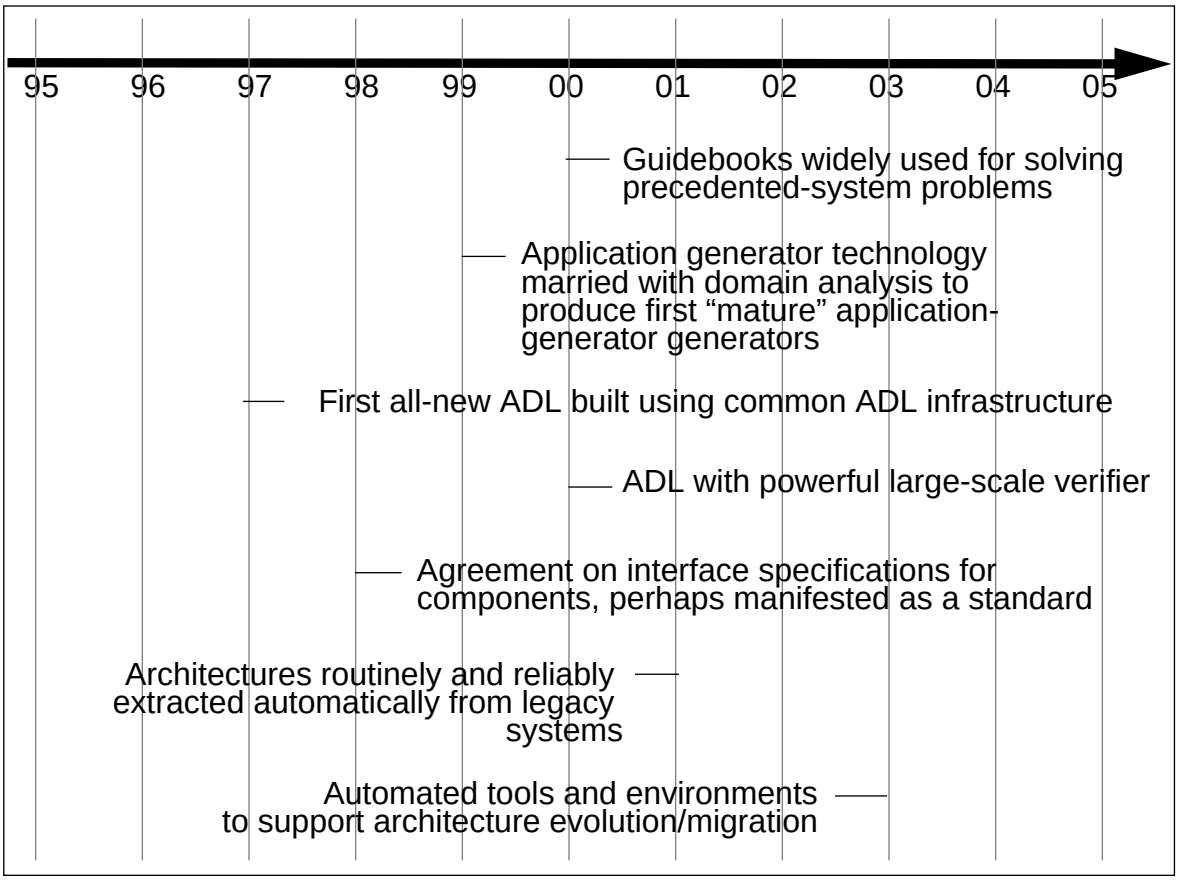

Figure 2: A possible timeline for coming attractions in architecture

6 Acknowledgments

Within the SEI, thanks go to Len Bass, Pat Donohoe, Rick Kazman, Mark Klein, and Craig Meyers for reviewing early versions of this paper; Kurt Wallnau and Michael Rissman provided especially thorough reviews.

Thoughtful external reviews were provided by Bob Balzer, Don Batory, David Garlan, Paul Kogut, Karl Lieberherr, Dewayne Perry, Mary Shaw, Walter Tichy, Bruce Weide, and Alex Wolf. David Garlan pointed out the limitations of the connector-interchange idea presented in Section 2. Sincere thanks go to all.

References

[Allen 94] Allen, R., & Garlan, D. “Formalizing Architectural Connection,” 71-80. *Proceedings of the 16th International Conference on Software Engineering*. Sorrento, Italy, May 16-21,1994. Los Alamitos: IEEE Computer Society Press,1994.

[Batory 92] Batory, D., & O’Malley, S. “The Design and Implementation of Hierarchical Software Systems with Reusable Components.” *ACM Transactions on Software Engineering and Methodology 1*, 4 (October 1992): 355-398.

[Batory 94] Batory, D.; Singhal, V.; Thomas, J.; Dasari, S.; Geraci, B.; & Sirkin, M. “The GenVoca Model of Software-System Generators.” *IEEE Software 11*, 5 (September 1994): 89-94.

[Beck 94] Beck, K. & Johnson, R. “Patterns Generate Architectures,” 139-149. *Proceedings of the 8th European Conference on Object-Oriented Programming*. Bologna, Italy, July 4-8, 1994. Berlin: Springer-Verlag, 1994.

[Berzins 95] Berzins, V. & Shing, R. “Summary of the ‘95 Monterey Workshop—Specification-Based Software Architectures,” 107-112. *Proceedings of the 1995 Monterey Workshop on Formal Methods and Software Architecture*. Monterey, Calif., September 12-14, 1995. Monterey: U.S. Naval Postgraduate School, 1996.

[Burch 90] Burch, J.; Clarke, E.; McMillan, K.; Dill, D.; & Hwang, L. “Symbolic Model Checking: 10^{20} States and Beyond,” 428-439*. Proceedings of the Fifth Annual IEEE Syhmposium on Logic in Computer Science*. Philadelphia, Pa., June 4-7, 1990. Los Alamitos: IEEE Computer Society Press,1990.

[Dijkstra 68] Dijkstra, E. W. “The Structure of the ‘T.H.E.’ Multiprogramming system.” *Communications of the ACM 26*, 1 (January 1983): 49-52.

[Fernandez 93] Fernandez, J. *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis* (CMU/SEI-93-TR-34, ADA279014). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993.

[Gamma 95] Gamma, E;, Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.

[Garlan 93] Garlan, D. & Shaw, M. “An Introduction to Software Architecture,” *Advances in Software Engineering and Knowledge Engineering* vol. I, River Edge, N.J.: World Scientific Publishing Company,1993.

[Garlan 94] Garlan, D.; Allen, R.; & Ockerbloom, J. “Exploiting Style in Architectural Design Environments.” *SIGSOFT Software Engineering Notes 19*, 5 (December 1994): 175-188.

[Garlan 95a] Garlan, D. “First International Workshop on Architectures for Software Systems.” *SIGSOFT Software Engineering Notes 20*, 3 (July 1995): 84-89.

[Garlan 95b] Garlan, D. & Perry, D. “Introduction to the Special Issue on Software Architecture.” *IEEE Transactions on Software Engineering 21,* 4 (April 1995): 269-274.

[Garlan 95c] Garlan, D. “Research Directions in Software Architecture.*” ACM Computing Surveys 27*, 2 (June 1995): 257-261.

[Inverardi 95] Inverardi, P. & Wolf, A. “Formal Specification and Analysis of Software Architectures Using the Chemical Machine Abstract Model.” *IEEE Transactions on Software Engineering 21*, 4 (April 1995): 373-386.

[Jackson 94a] Jackson, M. “Problems, Methods, and Specialization.” *IEEE Software 11*, 6 (November 1994): 57-62.

[Jackson 94b] Jackson, D. “Abstract Model Checking of Infinite Specifications,” 519-531*. Proceedings of The Second International Symposium of Formal Methods Europe*. Barcelona, Spain, October 24-28, 1994. Berlin: Springer-Verlag, 1994.

[Jahanian 86] Jahanian, F. & Mok, A. “Safety Analysis of Timing Properties in Real-Time Systems.” *IEEE Transactions on Software Engineering SE-12*, 9 (September 1986): 890-904.

[Jahanian 94] Jahanian, F. & Mok, A. “Modechart: A Specification Language for Real-Time Systems.” *IEEE Transactions on Software Engineering SE-20*, 12 (December 1994): 933-947.

[Kazman 94] Kazman, R.; Abowd, G.; Bass, L.; & Webb, M. “SAAM: A Method for Analyzing the Properties of Software Architectures,” 81-90*. Proceedings of the 16th International Conference on Software Engineering*. Sorrento, Italy, May 16-21, 1994. Los Alamitos: IEEE Computer Society Press, 1994.

[Landwehr 94] Landwehr, C.; Bull, A.; McDermott, J.; & Choi, W. “A Taxonomy of Computer Program Security Flaws.” *ACM Computing Surveys 26*, 3 (September 1994): 211-254.

[Lieberherr 96] Lieberherr, K. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. Boston, Mass.: PWS Publishing Company, 1996.

[Luckham 93] Luckham, D.; Vera, J.; Bryan, D.; Augustin, L.; & Belz, F. “Partial Orderings of Event Sets and Their Application to Prototyping Concurrent, Timed Systems.” *Journal of Systems and Software 21*, 3 (June 1993): 253-265.

[Moriconi 94] Moriconi, M. & Qian, X. “Correctness and Composition of Software Architectures.” *SIGSOFT Software Engineering Notes 19*, 5 (December 1994): 164-174.

[Murphy 94] Murphy, G.; Notkin, D.; & Sullivan, K. *Reflecting Source Code Relations in Higher-Level Models of Software Systems (*TR-94-09-93). Seattle, Wash.:University Of Washington, Department of Computer Science and Engineering, September 1994.

[Nierstrasz 95] Nierstrasz, O. & Meijler, T. D. “Research Directions in Software Composition.” *ACM Computing Surveys 27*, 2 (June 1995): 262-264.

[Object Management Group 95] Object Management Group, *Object Management Architecture Guide*. Third edition, New York, N.Y.: John Wiley & Sons, , 1995.

[Parnas 72] Parnas, D. “On the Criteria for Decomposing Systems into Modules.” *Communications of the ACM 15*, 12 (December 1972): 1053-1058.

[Parnas 74] Parnas, D. “On a ‘Buzzword’: Hierarchical Structure,” 335-342. *Programming Methodology*, Berlin, West Germany: Springer-Verlag, 1978.

[Parnas 76] Parnas, D “On the Design and Development of Program Families.” *IEEE Transactions on Software Engineering SE-2*, 1 (March 1976): 1-9.

[Perry 92] Perry, D. & Wolf, A. “Foundations for the Study of Software Architecture.” *ACM SIGSOFT Software Engineering Notes 17*, 4 (October 1992): 40-52.

[Scherlis 94] Scherlis, W. “Boundary and Path Manipulations on Abstract Data Types,” *IFIP Transactions A (Computer Science and Technology)* vol. A-5, Netherlands,1994.

[Scherlis 95] Scherlis, W. “Fluid Architecture and Semantics-Based Manipulations of Types,” Dagstuhl Workshop on Software Architectures, February 1995.

[Shaw 93] Shaw, M. *Software Architectures for Shared Information Systems* (CMU/SEI-93-TR-3, ADA266995). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1993.

[Shaw 94] Shaw, M. *Procedure Calls Are the Assembly Language of Software Interconnection; Connectors Deserve First-Class Status* (CMU/SEI-94-TR-02, ADA281026). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1994.

[Shaw 95] Shaw, M. “Making Choices: A Comparison of Styles for Software Architecture.” *IEEE Software 12*, 6 (November 1995): 27-41.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-96-TR-008

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-96-008

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/ENS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/ENS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F19628-95-C-0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Coming Attractions in Software Architecture

12. PERSONAL AUTHOR(S)
Paul C. Clements

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	January 1996	23

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
software architecture, architecture evaluation,
architecture case studies, application generators, architecture description languages, architecture recovery

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Software architecture is a field of study enjoying unprecedented growth and interest. This report identifies a set of promising lines of research related to software architecture and architecture-based system development that are expected to lead to advances available soon to practitioners. Some of the goals of software architecture are enumerated, and the investigatory efforts are structured according to work in the design or selection and creation, representation, evaluation and analysis, utilization, or legacy recovery of software architectures. Promising research is described in each area. These opinions are correlated with those of other experts in the field. Finally, a timeline for achieving some of

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/ENS (SEI)

ABSTRACT — continued from page one, block 19

the predicted results is offered.