The Coming-of-Age of Software Architecture Research

Mary Shaw
Institute for Software Research, International
Carnegie Mellon University
Pittsburgh PA 15213
1-412-268-2589
mary.shaw@cs.cmu.edu

Abstract

Over the past decade, software architecture research has emerged as the principled study of the overall structure of software systems, especially the relations among subsystems and components. From its roots in qualitative descriptions of useful system organizations, software architecture has matured to encompass broad explorations of notations, tools, and analysis techniques. Whereas initially the research area interpreted software practice, it now offers concrete guidance for complex software design and development.

We can understand the evolution and prospects of software architecture research by examining the research paradigms used to establish its results. These are, for the most part, the paradigms of software engineering. We advance our fundamental understanding by posing research questions of several kinds and applying appropriate research techniques, which differ from one type of problem to another, yield correspondingly different kinds of results, and require different methods of validation. Unfortunately, these paradigms are not recognized explicitly and are often not carried out correctly; indeed not all are consistently accepted as valid.

This retrospective on a decade-plus of software architecture research examines the maturation of the software architecture research area by tracing the types of research questions and techniques used at various stages. We will see how early qualitative results set the stage for later precision, formality, and automation and how results build up over time. This generates advice to the field and projections about future impact.

Keywords: Software architecture, research paradigms

1. Introduction

Software architecture is a relatively young area within software engineering. To assess its progress, maturity, and prospects, I will examine the growth of this area against the backdrop of normal growth and maturity in software engineering, paying special attention to the way we design and carry out research projects.

Redwine and Riddle [53] described the natural characteristics of maturing software technology. I elaborate their model by considering institutional progress as well. By comparing some of the highlights of software architecture to this model, I assess its progress toward maturity.

This maturity model also provides a framework for examining software architecture research in more detail. I examine the various kinds of research problems we investigate and the approaches we use to do so. Again, this provides a baseline for assessing the state of software architecture research.

Understanding the software architecture area in this context of normal progress suggests objectives and opportunities for the future.

2. How Technologies Mature

Redwine and Riddle [53] reviewed a number of software technologies to see how they develop and propagate. They found that it typically takes 15-20 years for a technology to be ready for popularization. They identify six typical phases:

- *Basic research*. Investigate basic ideas and concepts, put initial structure on the problem, frame critical research questions.
- *Concept formulation*. Circulate ideas informally, develop a research community, converge on a compatible set of ideas, publish solutions to specific subproblems.
- *Development and extension*. Make preliminary use of the technology, clarify underlying ideas, generalize the approach.

- *Internal enhancement and exploration*. Extend approach to another domain, use technology for real problems, stabilize technology, develop training materials, show value in results.
- *External enhancement and exploration*. Similar to internal, but involving a broader community of people who weren't developers, show substantial evidence of value and applicability.
- *Popularization*. Develop production-quality, supported versions of the technology, commercialize and market technology, expand user community

Redwine and Riddle presented timelines for several software technologies as they progressed through these phases. For abstract data types and information hiding, one of the important precursors of software architecture, they noted these transition points:

- formulation of "information hiding" [49] as the shift from basic research to concept formation,
- abstract data type languages [36][64] as the shift from concept formation to development and extension,
- major publications and frequent appearance of the concept in new programming languages (e.g., CLU [37]) as the shift to internal exploration, and
- propagation of abstract data types to other technologies such as the Affirm program verification system [23] as the shift to external exploration.

Their study ended in 1984, so they did not have the opportunity to note the influence of abstract data types on object-oriented programming techniques and on the incorporation of objects/classes in new programming languages.

As technologies mature, their institutional provisions for distributing results change. We begin with informal discussions among colleagues and progress to products in the marketplace. Along the way, we see preliminary results of the first two phases in position papers, workshops, and research conferences. As the ideas mature, results appear in conferences and then journals; larger conferences set up tracks featuring the technology, and eventually enough results are appearing to justify topical conferences. Books that synthesize multiple results help to move the technology through the middle phases. University courses, continuing education courses, and standards indicate the beginning of popularization.

Abstract data types emerged in a much smaller community than today's with somewhat different institutions. Nevertheless, we can see some of the institutional shifts. An informal group of researchers working on aspects of abstract data types met regularly once or twice a year in the mid to late 1970's in a format that resembles today's research workshops. SIGPlan Notices provided a forum similar to today's research workshop proceedings together with web-based distribution of reports [35][65]. Parnas led a sustained research program that began with testing information hiding ideas on small examples in the early 1970's [49] and culminated in a large-scale demonstration of applying the ideas to clean specification of the A7E avionics in the mid-1980s [50]. In the mid-1970s, the Strawman requirements for the Ada programming language design [15] clearly appealed to ideas from abstract data types. Abstract data type ideas were incorporated in a radical revision of the undergraduate data structures course that was classroom tested in the late 1970s and disseminated in a textbook [66]. The abstract data type research results did not enter widespread use directly; rather, they merged with inheritance results into object-oriented design and implementation, influencing the class/package constructs of object-oriented programming languages. The verification aspects of abstract data types have been slower to enter practice.

3. Maturation of software architecture

Software architecture is the principled study of the overall structure of software systems, especially the relations among subsystems and components. From its roots in qualitative descriptions of useful system organizations, software architecture has matured to encompass broad explorations of notations, tools, and analysis techniques. Whereas initially the research area interpreted software practice, it now offers concrete guidance for complex software design and development.

Software architecture research overlaps and interacts with work on software families, component-based reuse, software design, specific classes of components (e.g., COM), product line, and program analysis. It is not productive to attempt rigid separation among these areas; research can certainly contribute to more than one.

One way to see the growth of the field is to examine the rate at which earlier results serve as building blocks for subsequent results. A rough estimate is provided by citation counts for papers with "software architecture" in the title. For a sample of 2000 citations in the ResearchIndex database [47], virtually all of the cited papers were published in 1990 or later, and there were sharp increases in the numbers of citations for papers published after 1991 and again for papers published after 1994. The most widely-cited two dozen of these entries were published between 1990 and 1998. They include four books ([6][10][59][63], 1995 to 1998), eight papers dealing with architectures for particular domains ([11][12][13][24][32] [34][38][40], 1990 to 1995), seven papers presenting surveys or models for the field ([17][21][20][46][51] [57][58], 1992 to 1995), three formalizations ([1][2][45], 1993 to 1996),

and one paper each on an architecture description language [56] and reverse engineering [67]. Unsurprisingly, they generally represent the first three phases of development. Imperfect though this estimate may be, it still indicates very substantial growth since 1995 and a balance between exploration of specific problems and generalization and model development.

Here are some of the highlights of the field's development. The chronology is not as linear as the Redwine/Riddle model might suggest: different aspects of the field evolve at different rates; transitions between phases do not happen instantly; and publication dates lag the actual work by different amounts. Nevertheless, overall progress corresponds fairly well to their model.

3.1 Basic research

As long as complex software systems have been developed, their designers have described their structures with box-and-line diagrams and informal explanations. Good designers recognized stylistic commonalities among these structures and exploited the styles in ad hoc ways. These structures were sometimes referred to as architectures, but the knowledge of common styles was not systematically organized or taught.

In the late 1980s people began to explore the advantages of deliberately-designed specialized software structures for specific problems. Some of this work addressed software system structures for particular product lines or application domains, including oscilloscopes [14] and missile control [24][46]. Other work organized the informal knowledge about common software structures, or architectural styles, that can be used in a variety of problem domains. This work cataloged existing systems to identify common architectural styles such as pipe-filter, repository, implicit invocation, and cooperating processes [3][5][48][54]. These complementary lines of research led to models for explaining the architectural styles and to two widely cited papers in 1992 and 1993 that established the structure of the field [21][51].

3.2 Concept formulation

The basic models were elaborated and explored: through architecture description languages, early formalization, and classification. Most of this work took place in the mid-1990s, and it continues to the present as new ideas take shape.

Architecture description languages explored a variety of aspects of architecture. These languages included Aesop [18] (exploiting specific properties of styles), C2 [42] (exploring power of a particular event-based style), Darwin [41] (design and specification of dynamic distributed systems), Meta-H [7] (real-time avionics control), Rapide [39] (simulation and analysis of dynamic behavior expressed in posets), UniCon [56] (extensible set of connectors and styles, compilation to code), and Wright [3] (component interaction).

Formalizations developed in parallel with the language development. Sometimes this was integral to the language (Darwin, Rapide, Wright), and in other cases it was more independent, as the formalization of style [1][2] or formal analysis of inconsistency in a specific architectural model [60]. Recognition that multiple views must be reconciled in architectural analysis [31] helped to frame the requirements for formalism.

The early narrative catalogs of styles were expanded in taxonomies of styles [55] and of the elements that support those styles [30]. The common forms were cataloged and explained as patterns [10]. An early book [59] on these ideas set the stage for further development.

3.3 Development and extension

More recently, focus has been on unifying and refining initial results. The ACME architectural interchange language began with the goal of providing a framework to move information between architecture description languages [19]; it has grown to integrate other design, analysis and development tools.

Refinement of the taxonomies of architectural elements [44] and languages [43] has also continued.

The institutions of the area are also maturing. The Transactions on Software Engineering had a special issue on software architecture in 1995 [20]. The special "roadmap" track at the ICSE 2000 conference included software architecture [22] among its topics to survey. The current ICSE has four sessions on architectural topics, and of the 22 affiliated tutorials, 8 are related to UML and 6 others have clear architectural elements. A standalone conference, the Working IEEE/IFIP Conference on Software Architecture, will be held in August 2000, and one of the sponsors of that conference is a new IFIP working group.

3.4 Internal enhancement and exploration

Architectural styles are commonly used informally as design guides. The explicit attention to this aspect of design is increasing, and as a result we are gaining experience.

A few formal analyses of real system designs have been done as well. For example, architectural specification of the Higher Level Architecture for Distributed Simulation [4] was able to identify inconsistencies before implementation, thereby saving extensive redesign.

Books on the application of the research to practice [6][26] set the stage for external exploration.

3.5 External enhancement and exploration

Two unifications have matured enough to be useful outside their developer groups.

UML[8], under the leadership of Rational, has integrated a number of design notations and is developing a method for applying them systematically. The connection to concrete analysis and to code is not yet fleshed out, but the notations themselves have gained a wide audience.

The SEI's Architecture Tradeoff Analysis Method [29] supports analysis of the interaction among attributes as well as the attributes themselves.

3.6 Popularization

One of the hallmarks of a production-ready technology is good standards. Standards for particular component families (e.g., COM, CORBA) and interfaces (e.g., XML) exist, but they reflect component reuse interests as much as architectural interests. A recent IEEE standard [27] attempts to codify the current best practices and insights of both the systems and software engineering communities and make provisions for evolution.

3.7 Current status

It is fair to say that software architecture is well into the phase of development and extension, and that enhancement and exploration are beginning in earnest. Ideas and some tools are being exploited in practice, but the technologies are not yet mature. This is consistent with the shorter end of a 15-20 year maturity cycle. Research remains to be done, especially in the area of showing the range of utility of various styles, formalisms, design techniques, and tools. I turn now to a more detailed discussion of the research process itself.

4. Software engineering research paradigms

Software engineering research is often motivated by problems that arise in the production and use of real-world software. Such problems are often complex, and it's common to identify a research setting that is much simpler than the real problem but still captures some essential aspect. We can identify some common forms for these research settings, from narrative characterization of phenomena to rigorous analytic models. Researchers have a repertoire of approaches and methods; these yield a variety of research products, including system prototypes, procedural techniques, and models of several degrees of rigor.

4.1 Research settings

Software engineering research addresses several particular classes of problems. Polya [52] distinguished "problems to find" from "problems to prove", both in form and in the techniques used to obtain solutions. Similarly, Jackson distinguishes classes of software problems and develops a theory of "problem frames" to classify, analyze, and structure these problems [28]. Building on Shaw and Garlan's model of progressive codification [58] I make similar distinctions among different types research problems.

Technical ideas often begin as qualitative descriptions of problems or practice and gradually become more precise – and more powerful–as practical and formal knowledge grow in tandem. Thus, as some aspect of software development comes to be better understood, more powerful specification mechanisms become viable, and this in turn enables more powerful technology. Each of these stages of codification relies on a different kind of understanding and different approaches to the research.

General classes of research setting, the kinds of questions posed by each, and examples from abstract data types include:

Research Setting	**Sample Questions**	**Examples from abstract data types**
Feasibility	Is there an X, and what is it? Is it possible to accomplish X at all?	Is it possible to describe the relation among components of a software system [16]?
Characterization	What are the important characteristics of X? What is X like? What, exactly, do we mean by X? What are the varieties of X, and how are they related?	What is the important information to share and to hide about a component [49]?
Method/-Means	How can we accomplish X? What is a better way to accomplish X? How can I automate doing X?	How can we incorporate abstract data types in a programming language [37][64]?
Generalization	Is X always true of Y? Given X, what will Y be?	What is a formal relation between the specification of an abstract data type and its implementation [25]?
Selection	How do I decide between X and Y?	What organization should I choose for the user interface component of a system [33]?

Examples from software architecture include:

- *Feasibility.* Early architecture description languages showed that it is possible to specify component interactions [3] or to compile from a specification, including generation of glue code for connectors [56].

- *Characterization*. The early models for the structure of the field [21][51] characterized both the phenomena and the terms in which they could be explained.
- *Method/means.* Many of the papers that investigate architectures for particular classes of systems [14][24][50] are finding ways, and improving ways, to design and develop those systems.
- *Generalization*. Taxonomies of elements and of styles [43][44][55][30] built on and generalized the early characterizations. Patterns identify the conceptual units that are regularly used in design [10].
- *Selection*. Decision support for choosing a system structure [33]. The Architecture Tradeoff Analysis Method [29] helps with selection by supporting analysis of the interaction among attributes.

4.2 Research approaches, methods, and products

Software engineering researchers approach problems in a variety of ways. I distinguish them by the tangible results of the research project.

Research Product	Research Approach or Method	Examples
Qualitative or descriptive model	Organize & report interesting observations about the world. Create & defend generalizations from real examples. Structure a problem area; formulate the right questions. Do a careful analysis of a system or its development.	Early architectural models [21][51], architectural patterns[10],
Technique	Invent new ways to do some tasks, including procedures and implementation techniques. Develop a technique to choose among alternatives	Product line and domain-specific software architectures [14] [24], UML to support object-oriented design[8]
System	Embody result in a system, using the system development as both source of insight and carrier of results	Architecture description languages, especially ACME
Empirical predictive model	Develop predictive models from observed data	
Analytic model	Develop structural (quantitative or symbolic) models that permit formal analysis	HLA specification, COM inconsistency analysis

4.3 Validation techniques

Results alone do not suffice. Although this step is often neglected, the results must be validated in some way to show that they satisfy the requirement posed by the research setting. Depending on the way the solution is created and validated, we may have more or less confidence in its correctness, adequacy, or generality. As ideas mature, more rigor and detailed analysis is required to advance our knowledge.

When describing a result, it is helpful to be explicit about what degree of precision and rigor it purports to achieve. Indeed, Brooks [9] proposes recognizing three kinds of results, with individual criteria for quality:

- *Findings.* well-established scientific truths – judged by truthfulness and rigor
- *Observations.* reports on actual phenomena – judged by interestingness
- *Rules-of-thumb*. generalizations, signed by an author (but perhaps not fully supported by data) – judged by usefulness

with freshness as criterion for all.

Some argue for a narrower standard that would require quantitative data on experimental results for most research [61][62][68]. I take a more nuanced position, that different types of results have value, and that more rigorous results emerge only over time, either through cumulative evidence or by building rigorous experiments on a base of more informal experience. In other words, as engineers we do the best we can with the available information, but in doing so we must consider carefully how much to trust each result. Researchers do, of course, have an obligation to explain clearly how and to what extent their results support their hypotheses and solve the problems that they claim to solve – that is, to provide validation of their results.

Good validation entails not only showing that the specific product of the research satisfies the idealized problem of the research setting, but also that the result helps to solve the original motivating problem. Even in papers that validate the result against the research setting, this is often omitted. As in most of software engineering, excellent examples are rare and inadequate discussion is altogether too common.

Formal research often provides verification of its results; Sullivan's formal analysis of an inconsistency in the COM specification [60] not only provides a good example of verification, but also of carrying the result back to the original problems.

As noted above, early architecture description languages showed that it is possible to specify component interactions [3] or to compile from a specification, including generation of glue code for connectors [56]. To the extent

that these projects addressed sheer feasibility questions, the implementation of a working system constitutes validation.

The early models for the structure of the field [21][51] were originally supported by persuasion. As time has passed and those models have been built upon and refined, the cumulative validation includes experience.

Some of the techniques that have been used convincingly for validating one or another kind of software engineering research result are

Technique	Character of validation
Persuasion	I have thought hard about this, and I believe that…
Technique	…if you do it the following way, then…
Design	…a system constructed like this would…
Example	…walking through this example shows how my idea works
Implementation	Here is a prototype of a system that…
System	…exists in code or other concrete form
Technique	…is represented as a set of procedures
Evaluation	Given these criteria, here's how an object rates…
Descriptive model	…in a comparison of many objects
Qualitative model	…by making subjective judgments against a checklist
Empirical quantitative model	…by counting or measuring something
Analysis	Given the facts, these consequences…
Analytic formal model	…are rigorous, usually symbolic, in the form of derivation and proof
Empirical predictive model	are predicted by the model in a controlled situation (usually with statistical analysis)
Experience	I evaluate these results based on my experience and observations about the use of the result in actual practice and report my conclusions in the form of
Qualitative or descriptive model	…prose narrative
Decision criteria	…comparison of systems in actual use
Empirical predictive model	…data on use in practice, usually with statistical analysis

Brooks' "observations" and "rules of thumb" correspond well to qualitative or descriptive models.

Two kinds of inadequate arguments appear entirely too often in the software engineering literature. One sets out to improve current practice, proposes a new technique, implements a tool to support the technique, and fails to collect any evidence comparing the new technique and tool to the practice it purports to improve. The second proposes a new technique, applies it to a toy example, and claims a contribution; this form is particularly unsatisfactory when the example is only tenuously related to a practical problem.

4.4 Selecting a research strategy

The research strategy for a software engineering problem identifies a research setting that addresses a fundamental issue in the problem and a matching approach and validation technique. Different approaches are appropriate for different settings, and similarly different validation techniques are appropriate for different types of results. Judicious selection of compatible settings, approaches, research products, and validation techniques is a critical part of designing a research project.

For example, if there was at the outset a serious question about whether something could be achieved at all, simply exhibiting an implementation is validation enough. However, if the question is how to improve on current practice, the same implementation would probably be completely unsatisfactory as the sole evidence. In other words, "Look, it works!" is sufficient proof only if the original question was whether it could be done at all.

As ideas mature into practical technologies, the character of the open questions changes. Whereas clear articulation of issues may be a significant contribution at the outset, the questions as a technology enters practice may involve formal analysis and evaluation of effectiveness relative to other technologies. Since the essential questions change, so will the idealized research settings, the most suitable approaches and research products, and the appropriate validation techniques.

I offer these observations about research techniques and strategies in the sense of Brooks' rules of thumb. They are open to refinement, and their usefulness must be judged by whether they guide researchers to more careful design of their projects and their reports of results.

5. What Next?

We see that software architecture is reaching the point of growing from its adolescence in research laboratories to the responsibilities of maturity. This brings with it additional responsibility to show not just that ideas

are promising (a sufficient grounds to continue research) but also that they are effective (a necessary grounds to move into practice).

As a result, software architects must not be content with simply doing more research in the style of the past decade. Certainly there are new ideas yet to be explored in that form, but we must also attend to making existing results more robust, more rigorously understood, and more ready to move into application.

There is a lesson here for all of software engineering as well. Notwithstanding over 30 years of existence, software engineering does not yet have a widely-recognized and widely-appreciated set of research paradigms in the way that other parts of computer science do. That is, we don't recognize what our research strategies are and how they establish their results. Poor external understanding leads to lack of appreciation and respect. Poor internal understanding leads to poor execution, especially of validation, and poor appreciation of how much to expect from a project or result. There may be also be secondary effects on the way we choose what problems to work on at all.

So there is here a challenge to the whole of software engineering to think through our research models more carefully, to learn how to determine how much trust should be put in our results, and to take the effort to validate the work of both individual research papers and of long-term projects.

6. Acknowledgments

I appreciate the patience of my colleagues in the Institute for Software Research International at Carnegie Mellon in helping me work through these ideas, especially David .Garlan and the students in the spring 2000 course in which I started serious exploration of the research paradigms of software engineering. I appreciate David Notkin's evening open mike sessions at the FSE conference and his willingness to let me raise related questions repeatedly. I especially appreciate the invitation to present a keynote talk and paper at ICSE 2000, which presents an ideal forum in which to raise these issues for the software engineering community generally as well as the software architecture community.

7. References

[1] Gregory Abowd, Robert Allen, David Garlan. Using Style to Understand Descriptions of Software Architecture, *Proc. 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1993

[2] Gregory Abowd, Robert Allen, and David Garlan. Formalizing style to understand descriptions of software architecture. *ACM Tr on Software Engineering and Methodology*, 1995.

[3] Robert Allen and David Garlan. Formalizing architectural connection. *Proc 16th International Conference on Software Engineering*, May 1994, pp. 71-80.

[4] Robert Allen, David Garlan, and James Ivers. Formal modeling and analysis of the HLA component integration standard. Proc 6th Intl Symposium on the Foundations of Software Engineering, FSE-6, November 1998.

[5] Gregory Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, vol 23 no 1, March 1991, pp. 49-90.

[6] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998

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

[8] Grady Booch. *UML Users Guide*. Addison Wesley, Chicago, Il, 1998.

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

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996

[11] B. Chapman, P. Mehrotra,, J. Van Rosendale, and H. Zima. A software architecture for multidisciplinary applications: Integrating task and data parallelism. *Tech. Rep. 94-18, ICASE*, NASA Langley Research Center, Hampton, VA, Mar. 1994.

[12] G. Chiola: GreatSPN 1.5 Software Architecture; *Proc. 5th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Torino, 13-15 Feb. 1991.

[13] J. Cremer, J. Kearney, Y. Papelis, and R. Romano. The software architecture for scenario control in the Iowa driving simulator. *Proc Conference on Computer Generated Forces and Behavioral Representation*

[14] Norman Delisle and David Garlan. Formally specifying electronic instruments. *Proc. Fifth International Workshop on Software Specification and Design*, May 1989.

[15] US Department of Defense. Strawman Requirements for Higher Order Programming Languages., 1975.

[16] Frank DeRemer and Hans H. Kron. Programming-in the -Large Versus Programming-in-the-Small. *IEEE Tr on Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.

[17] David Garlan. Research direction in software architecture. *ACM Computing Surveys*, vol 27 no 2, 1995, pp. :257-261.

[18] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. *Proc SIGSOFT '94: 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994, pp. 170-185.

[19] David Garlan, Robert Monroe, and David Wile. Acme: An architecture description interchange language. *Proc CASCON '97*, November 1997, pp.169-183.

[20] David Garlan and Dewayne Perry. Introduction to the Special Issue on Software Architecture, *IEEE Tr on Software Engineering*, vol 21 no 4, April 1995, pp 269-274.

[21] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Co., 1993.

[22] David Garlan. Software architecture: a roadmap. *The Future of Software Engineering 2000, Proceedings 22nd International Conference on Software Engineering*, ACM Press 2000

[23] S.L. Gerhart, D.R. Musser, D.H. Thompson, D.A. Baker, R.L. Bates, R.W. Erickson, R.L. London, D.G. Taylor and D.S. Wile. An Overview of AFFIRM: A Specification and Verification System. *Information Processing 80*, S. H. Lavington (Ed.), October, 1980, pp. 343-348.

[24] Mark Goodwin and Marty Kushner, Domain Analysis for the Avionics Domain Architecture Generation Environment of Domain Specific Software Architecture. *ADAGE*-IBM-92-11.

[25] C.A.R. Hoare. Proofs of Correctness of Data Representations. *Acta Informatica*, Vol. 1, 1972, pp. 271-281.

[26] C. Hofmeister, R. Nord, and D. Soni. *Applied software architecture.* Addison-Wesley, 1999

[27] IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE, 2000*.

[28] Michael Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[29] R. Kazman, M. Barbacci, M. Klein, S.J. Carrière, Experience with Performing Architecture Tradeoff Analysis, *Proc ICSE '99*, May 1999, 54-63.

[30] R. Kazman, P. Clements, L. Bass, and G. Abowd.. Classifying Architectural Elements as a Foundation for Mechanism Matching, Proc. *COMPSAC '97 International Computer Software and Applications Conference*, August 1997, pp. 1417.

[31] P. Kruchten. The 4+1 View Model of Software Architecture. *IEEE Software* (Nov. 1995): 42-50.

[32] S. Kumar and E. H. Spafford. A software architecture to support misuse intrusion detection. *Proc 18th National Information Security Conference*, 1995, pp 194-204.

[33] 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, November 1990.

[34] W.H. Leung, T.J. Baumgartner, Y.H. Hwang, M.J. Morgan, and S.C. Tu. A Software Architecture for Workstations Supporting Multimedia Conferencing in Packet Switching Networks. *IEEE Journal on Selected Areas in Communications*, vol.8, no.3, April 1990, pp.380-390.

[35] B. Liskov and S. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, April 1974.

[36] B. Liskov and S. Zilles. Specification techniques for Data Abstractions. *IEEE Tr on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 7-19.

[37] B. Liskov et al.: *CLU Reference Manual*, Springer-Verlag, 1981, 190p.

[38] C. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Journal of Real-Time Systems*, vol 4 no 1, March 1992, pp. 37-53.

[39] D.C. Luckham, L.M. Augustin, J.J. Kenny, J. Veera, D. Bryan, W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Tr on Software Engineering* vol 21 no 4, April 1995, pp 336-355.

[40] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz, NPSNET: A Network Software Architecture for Large Scale Virtual Environments. *Presence*, vol. 3, no. 4. Fall 1994.

[41] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Proc 5th European Software Engineering Conference, ESEC '95*, September 1995

[42] N. Medvidovic, P. Oreizy, J.E. Robbins, R.N. Taylor. Using object-oriented typing to support architectural design in the C2 style. *Proc 1st Working IFIP Conference on Software Architecture, WICSA1*, February 1999.

[43] N. Medvidovic and R.N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. *Proc 6th European Software Engineering Conference*, Lecture Notes in Computer Science 1301, pages 60--76, September 1997.

[44] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. *Proc. International Conference on Software Engineering, 2000*.

[45] D. Le Metayer. Software Architecture Styles as Graph Grammers. *Proc 4th ACM SIGSOFT Symposium on the*

Foundations of Software Engineering, November 1996, pp 15--23.

[46] E. Mettala and M. Graham (eds.), The Domain-Specific Software Architecture Program, *Technical Report CMU/SEI-92-SR-9*, Software Engineering Institute, Carnegie Mellon University, 1992.

[47] NEC. ResearchIndex: The NECI Scientific Literature Digital Library, http://citeseer.nj.nec.com/cs

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

[49] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, vol. 15, no. 12, December 1972, pp. 1053-1058.

[50] D.L. Parnas, P.C. Clements, and D.M. Weiss. The Modular Structure of Complex Systems. *IEEE Tr on Software Engineering*, Vol. SE-11, No. 3, March 1985, pp. 259-266.

[51] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17:pp. 40-52, October 1992.

[52] George Polya. *How to Solve It*. Princeton University Press, reissue edition 1971.

[53] Samuel Redwine and William Riddle. Software technology maturation. *Proc 8th International Conference on Software Engineering*, May 1985, pp. 189-200..

[54] Mary Shaw. Toward Higher-Level Abstractions for Software Systems. Proc. Tercer Simposio Internacional del Conocimiento y su Ingerieria, October 1988 (printed by Rank Xerox) (invited), pp.55-61. Revised as Larger-Scale Systems Require Higher-Level Abstractions, *Proc. 5th Int'l Workshop on Software Specification and Design*, Pittsburgh, May 1989, pp.143-146.

[55] Mary Shaw and Paul. Clements. A Field Guide to Boxology: Preliminary Classication of Architectural Styles for Software Systems. *COMPSAC '97 International Computer Software and Applications Conference*, August 1997, pp 6-13,

[56] M. Shaw, R. DeLine, V. Klein, T.L. Ross, D.M. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Tr on Software Engineering*, Vol. 21, No 4, April 95.

[57] Mary Shaw and David Garlan. Characteristics of Higher-level Languages for Software Architecture. *Technical Report CMU-CS-94-210*. Also appears as CMU/SEI-94-TR-23, ESC-TR-94-023.

[58] Mary Shaw and David Garlan. Formulations and Formalisms in Software Architecture. Jan van Leeuwen, ed., *Computer Science Today: Recent Trends and Developments*, Lecture Notes in Computer Science, Volume 1000, pp. 307-323, Springer-Verlag, 1995.

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

[60] Kevin Sullivan, M. Marchukov and D. Socha. Analysis of a conflict between interface negotiation and aggregation in Microsoft's component object model. *IEEE Tr on Software Engineering*, July/August, 1999.

[61] W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: A quantitative study. *J. Systems Software* Vol. 28, No. 1, 1995, pp. 9-18.

[62] Walter F. Tichy. Should computer scientists experiment more? 16 reasons to avoid experimentation. *IEEE Computer*, Vol. 31, No. 5, May 1998.

[63] Kim Walden and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice Hall, 1995.

[64] W. A. Wulf and R. L. London and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Tr on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 253-265.

[65] W. A. Wulf and M. Shaw: Global variables considered harmful, *SIGPLAN Notices*, vol. 8 no 2 Feb. 1973, pp 28-34.

[66] Wm. A. Wulf, Mary Shaw, Paul N. Hilfinger, and Lawrence Flon, *Fundamental Structures of Computer Science* Addison-Wesley, 1981

[67] A.S. Yeh, D. R. Harris, and M. P. Chase. Manipulating recovered software architecture views. *Proc International Conference on Software Engineering*, 1997, pp 184--194.

[68] M. Zelkowitz and D. Wallace. Experimental validation in software engineering. *Information and Software Technology*, November 1997.

When people in the software industry talk about â€œarchitectureâ€, they refer to a hazily defined notion of the most important aspects of the internal design of a software system. A good architecture is important, otherwise it becomes slower and more expensive to add new capabilities in the future. Like many in the software world, Iâ€™ve long been wary of the term â€œarchitectureâ€ as it often suggests a separation from programming and an unhealthy dose of pomposity. But I resolve my concern by emphasizing that good architecture is something that supports its own evolution, and is deeply intertwined w Weâ€™re on the cusp of another Golden Age that will significantly improve cost, performance, energy, and security. These architecture challenges are even harder given that weâ€™ve lost the exponentially increasing resources provided by Dennard scaling and Mooreâ€™s law. Weâ€™ve identified areas that are critical to this new ageÂ Thus far, architects have been asked for little beyond page-level protection and supporting virtual machines. The very definition of computer architecture ignores timing, yet Spectre shows that attacks that can determine timing of operations can leak supposedly protected data. Itâ€™s time for architects to redefine computer architecture and treat security as a first class citizen to protect data from timing attacks, or at worst reduce information leaks to a trickle.