Research Brief on

# Composable Software Systems

*David Garlan, Daniel Jackson, Mary Shaw, and Jeannette Wing*

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
February 21, 1996

http://www.cs.cmu.edu/~Compose/

## 1 Theme: Putting Systems Together

Software techniques have developed enormously over the past 30-40 years, from primitive machine languages to sophisticated programming languages and tools for system configuration. We are working on the next great leap, shifting focus from programming-at-the-module level to programming-at-the-system level. In the former activity we build programs in terms of procedures and abstract data types using simple module interconnection as a way to compose modules; in the latter activity, we build systems in terms of more sophisticated components, often entire systems themselves, and compose them with more sophisticated abstractions for connection or interaction.

What makes the construction of composable systems different from programming?

1. We are liberating ourselves from thinking of the task as merely programming. We are not just building a program, we are building a system, where in the most general sense a system has not just software but hardware components as well.

2. Our units of manipulation are *components* and *connectors*. They may be primitive or composite. We describe components with *interface* specifications; connectors, *protocol* specifications. Our innovation is to make connectors first-class entities, just as components are, in a system. Program modules like procedures are examples of primitive components. Procedure call is one example of a primitive connector between two components. Designers use much richer interactions than procedure call, and so must our connectors. We want to support composite components and connectors whose properties, via their interface and protocol specifications, are as understandable as their constituents.

3. We want to provide ways to talk precisely about common patterns of system organization structures. This allows us to provide larger abstractions for system structuring.

It also allows us to exploit special properties of particular composition idioms for analysis, design guidance, and efficient implementation.

Current composition technology falls short both for synthesis and analysis. On the synthesis side, we see a proliferation of specialized solutions, but no systematic support for sharing expertise or for resolving differences in interaction assumptions. On the analysis side, we need component-level analysis, but we have a poor understanding of interface abstractions, and our analysis techniques are too weak.

## 2 Research Goals

The research program at Carnegie Mellon in Composable Software Systems has two primary goals:

1. To provide a scientific and engineering basis for design, analysis, and composition of flexible systems from complex building blocks;

2. To provide languages, tools, environments, and techniques to support (1).

Research in the 1970's gave us a *science of programming*, providing the basis to formalize the meaning of a program in terms of the meanings of its constituents. We identified composition mechanisms such as sequential composition (;) for sequential programming and parallel composition (||) for parallel programming. We developed reasoning aids like loop invariants and termination functions to help programmers design and verify their code. Undergraduates now routinely learn how to perform formal reasoning and analysis of (sequential) programs made up of compositions of primitive constituents; they learn to make well-reasoned selections of algorithms and data structures to meet the needs of the problem at hand. We now have tools and programming environments that support both the day-to-day programming process and the less frequent formal reasoning process.

Achieving our first goal would give the community a *science of composing systems* . It would also establish models and principles for making pragmatic design decisions. Achieving our second goal would give the community tangible ways to compose systems in a systematic, rather than ad-hoc, way.

## 3 Specific Research Activities at Carnegie Mellon

A system consists of components, described by interfaces, and connectors, described by protocols. Components communicate with each other through connectors. In Figure 1 component A communicates with C using protocol P; B with C using Q; A and B do not directly communicate. A and B could both be clients communicating through remote procedure call to server C; A, B, and C could be a filters connected through pipes P and Q. A could be a user process making synchronous calls to the file system manager C which asynchronously reads and writes from disk B.

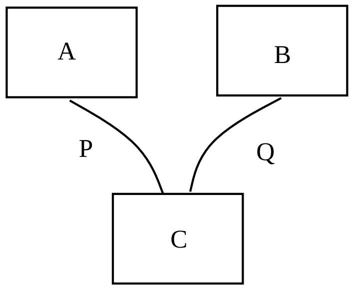


Figure 1: Components and Connectors

Though we are all interested in understanding whole diagrams of the kind in Figure 1 and their multitude of interpretations and applications, each of us is also specifically focusing on various aspects of these diagrams. Mary Shaw and David Garlan both focus on the ways to configure the components and connectors (and on the styles and meanings of the configurations). David Garlan also focuses on connectors and their protocols; Jeannette Wing, on components and their interfaces; Daniel Jackson, on partial and whole configurations where components and connectors may themselves only be *partially* specified (imagine dotted, rather than solid, lines for the boxes and lines in Figure 1). Mary Shaw is also interested in how designers make choices among structural alternatives.

There are also four themes that cut cross our individual research projects and interests: software architectures (Garlan, Shaw), formal methods (Garlan, Jackson, Wing), tractable software analysis (Jackson), and design guidance (Shaw).

In the following sections we give more details of our individual research interests and projects and a list of relevant publications. Please feel free to contact any of us through e-mail at *firstname.lastname*@cs.cmu.edu for more information.

### 3.1 Mary Shaw

The *architecture of a software system* defines that system in terms of components and interactions, or connections, among those components. In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system. It addresses system-level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and their connectors. architectural definitions can be composed to define larger systems. Elements are defined independently so they can be re-used in different contexts. The architecture establishes specifications for individual elements that may be written in a conventional programming language. A number of commonly-used patterns, or idioms, are in widespread informal use; these architectural styles can be captured as general templates for families of related systems. This holds particular promise for domain-specific systems.

My current primary concerns are identifying and supporting architectural abstractions, localizing and codifying the ways components interact, and distinguishing among the various packagings of components that require different forms of interaction. A major thrust of the research is identifying suitable abstractions for architectural styles and connectors, together with language to describe them. My focus is largely pragmatic. The first goal of the work has been to identify, classify, and support a variety of components and their connections. A prototype tool now supports eight component types and six connector types. Over time, the notation will be refined and a formal base will be developed.

Research questions of interest include:

- How can software system structure be characterized? What properties matter?
- What abstractions are useful for structures, systems, and subsystems? How are the abstractions related to each other?
- What textual and graphical notations for architecture are useful to system designers?
- How do you cope when you must build a system with parts that make incompatible assumptions about their mutual interaction?
- What kinds of checking and analysis we support?
- How can we support good architectural design decisions, especially choices among architectural styles?

In addition to developing notations, models, and tools, I actively disseminate the results. These activities center on developing and presenting educational materials on software architecture. This has to date resulted in a course, a tutorial presented at ICSE 93, SIGSOFT 93, SIGSOFT 94, and the 1996 CMU summer school, and a videotape for distribution. I am also active in promoting the ideas in professional forums. Currently, the CMU software architecture group is organizing a set of "model problems" in the interest of sharing common examples within the research community. It is available from the http://www.cs.cmu.edu/~ModProb/ web address.

## References

[Shaw and Garlan 96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Engineering Discipline*. Prentice Hall, to appear March 1996.

[Shaw et. al. 96] Mary Shaw, Robert DeLine, Gregory Zelesnik. "Abstractions and Implementations for Architectural Connections." International Conference on Configurable Distributed Systems, May 1996 (to appear).

[Shaw 96a] Mary Shaw. "Truth vs Knowledge: The Difference Between What a Component Does and What We *Know* It Does." *Proceedings of the 8th International Workshop on Software Specification and Design*, March 1996 (to appear).

[Shaw 96b] Mary Shaw. "Some Patterns for Software Architecture." *Results from Second Annual Conference on Pattern Languages of Programming*, September 1995 (to be published by Prentice-Hall, 1996).

[Shaw 96c] Mary Shaw. "Software Architectures for Shared Information Systems." *Mind Matters: Contributions to Cognitive and Computer Science in Honor of Allen Newell.* Lawrence Erlbaum, to appear 1996.

[Shaw 95] Mary Shaw. "Making Choices: A Comparison of Styles for Software Architecture." *IEEE Software*, special issue on software architecture, vol 12, no 6, November 1995, pp.27-41.

[Shaw et al 95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David Young, and Gregory Zelesnik. "Abstractions for Software Architecture and Tools to Support Them." *IEEE Transactions on Software Engineering*, vol 21, no 4, April 1995, pp. 314-335.

[Lane 90] Thomas G. Lane, "Studying Software Architecture Through Design Spaces and Rules," Technical Report CMU/SEI-90-TR18 ESD-90-TR-219, Carnegie Mellon University, September 1990.

## 3.2 David Garlan

Good architectural design has always been a major factor in determining the success of a complex software system. However, while there are many useful compositional paradigms (pipelines, layered systems, client-server organizations etc.), they are typically understood only in an idiomatic way and applied in an ad hoc fashion. Consequently, software system designers have been unable to fully exploit commonalities in system architectures, make principled choices among design alternatives, or teach their craft to others. To address these issues my research has three complementary components:

1. **Architectural Specification and Analysis:** A stumbling block to effective architectural design is our current inability to characterize software architectures precisely and to reason about their properties. I am attacking this problem by developing formal models of software architectures to provide precise, abstract characterizations of their properties and their dimensions of specialization. First, I am developing specification languages for architectural design. This work focuses on the treatment of connectors as first class semantic entities. Second, I am developing a formal basis for specification of architectural *style* – that is, common idioms and patterns of architectural decomposition. This work is concerned with both formal characterization of specific styles and also a general framework for such specification. Of particular interest are architectural styles based on event broadcast, a common structuring mechanism for composable systems.

2. **Tools and Environments:** Given precise models of software architectures it becomes possible to support architectural design with new tools and environments. I have begun work on an architectural environment toolkit, called Aesop. When complete it will allow a designer to define specialized architectural styles and then quickly construct environments to support design using those styles. A primary challenge of this research is to develop an open framework that accommodates not only architectural descriptions built with our tools, but also artifacts (code, documentation, reuse libraries, etc.) and tools (compilers, design checkers, etc.) built by others. A second

challenge is to find appropriate declarative mechanisms that make it easy to describe new architectural styles and to fashion new styles by augmenting or combining existing ones.

3. **Education of Software Architects:** Mary Shaw and I have developed material for courses in software architectures. One aspect of this is a full semester course for upper level undergraduates and masters students. A second aspect is the creation of materials for a half-day tutorial on software architecture, which we have delivered at several major conferences. A third aspect is a book, which we are writing as an introduction to the emerging field of software architecture.

## References


[Garlan and Delisle 95] Garlan, D., and Delisle, N. "Formal Specification of an Architecture for a Family of Instrumentation Systems." Chapter 4 of "Applications of Formal Methods," Prentice Hall, Hemel Hempstead, September 1995.

[Garlan et al. 95] Garlan, D., Allen, R., and Ockerbloom, J. "Architectural Mismatch, or, Why it's hard to build systems out of existing parts," in Proceedings of the 17th International Conference on Software Engineering, 1995.

[Garlan et al. 94] Garlan, D., Allen, R., and Ockerbloom, J. "Exploiting Style in Architectural Design Environments," in Proceedings of SIGSOFT'94: Foundations of Software Engineering, 1994.

[Allen and Garlan 94] Allen, R. and Garlan, D. "Formalizing Architectural Connection." in Proceedings of the 16th International Conference on Software Engineering, 1994.

[Abowd et al. 93] Abowd, G., Allen, R., and Garlan, D. "Using Style to Give Meaning to Software Architecture." in Proceedings of SIGSOFT'93: Foundations of Software Engineering. Software Engineering Notes 118(3) ACM Press, 1993.

[Garlan and Shaw 93] Garlan, D. and Shaw, M. "An Introduction to Software Architecture," in Advances in Software Engineering and Knowledge Engineering, Volume I, edited by V. Ambriola and G. Tortora. World Scientific Publishing Company, 1993.

[Garlan et al. 92] Garlan, D., Shaw, M., Okasaki, C., Scott, C., and Swonger, R. "Experience with a Course on Architectures for Software Systems," in Proceedings of the Sixth SEI Conference on Software Engineering Education. Springer Verlag, LNCS 376, 1992.


### 3.3 Jeannette Wing

I am interested in the *specification of interfaces*. Interfaces should contain semantic information, not just syntactic (like types) and structural information (like imports/exports). Semantic information includes not just the functional behavior of a component, but characterizations of "non-functional" behavior as well, e.g., reliability, performance, security. The scientific challenge is to understand how different models for describing different behavioral characteristics can be combined in meaningful ways. The engineering challenge is to define an interface definition language that is sufficiently expressive to describe a component's semantic information, yet at the same time be restrictive enough so that relevant semantic checks can be performed automatically or semi-automatically.

Given interface specifications of components and protocol specifications of connectors, I am also interested in how to achieve *system interoperability*. An interoperability problem arises if components do not speak the same language when trying to communicate. There could be an interface mis-match or protocol mis-match. To solve this problem, we must first be able to detect when there is a mis-match, then decide whether to resolve it, and if so, how. The scientific challenge here is to formulate a model to understand and characterize the interoperability problem better. One engineering challenge is devising solutions for resolving semantic mis-matches.

My specific domain of application is *distributed systems*. The interface definition language mentioned above should be appropriate for describing components in a distributed system. The presence of concurrency and faults makes the underlying model more complex than that for sequential systems. Yet these are exactly the complexities systems builders have difficulty formalizing and reasoning about. Theoretical research in models and logics have much to offer; however, they tend not to be accessible to practitioners. Thus, the language I want to design must be usable by real systems builders.

One of the hot trends in distributed computing today is the growth of large-scaled information systems like the World Wide Web. The President's vision of the National Information Infrastructure is like a WWW for the general public. In this context, I am interested in the problem of *search*. I would like to exploit the structural and semantic information of the components being searched to enhance and expedite the search process. Ideally, a query could be akin to an interface specification; retrieval would be based on a semantic match between the query and the interfaces of desired components.

## References


[Wing and Vaziri-Farahani 95] J.M. Wing and M. Vaziri-Farahani, "Model Checking Software Systems: A Case Study," in the *Proceedings of SIGSOFT Foundations of Software Engineering*, October 1995. Also CMU-CS-95-128, March 1995.

[Zaremski and Wing 95] A.M. Zaremski and J.M. Wing, "Specification Matching of Software Components," in the *Proceedings of SIGSOFT Foundations of Software Engineering*, October 1995. Recommended by program committee for submission to *ACM Transactions on Software Engineering and Methodology*. Also CMU-CS-95-127, March 1995.

[Mummert et al. 94] L. Mummert, J.M. Wing, and M. Satyanarayanan, "Using Belief to Reason about Cache Coherence," to appear in Symposium on Principles of Distributed Computing, August 1994.

[Liskov and Wing 94] B.H. Liskov and J.M. Wing, "A Behavioral Notion of Subtyping," ACM Transactions on Programming Languages and Systems, Volume 16, Number 6, November 1994, pp. 1811-1841. Also CMU-CS-93-187 (supersedes CMU-CS-93-149 and CMU-CS-92-220).

[Haines et al. 94] N. Haines, D. Kindred, J.G. Morrisett, S.M. Nettles, and J.M. Wing, "Composing First-Class Transactions," ACM Transactions on Programming Languages and Systems, Short Communications, Volume 16, Number 6, November 1994, pp. 1719-1736. Also Technical Report CMU-CS-93-202 under title "Tinkertoy Transactions."

[Zaremski and Wing 93] A.M. Zaremski and J.M. Wing, "Signature Matching: A Tool for Software Libraries," ACM Transactions on Software Engineering and Methodology, Volume 4, Number 2, April 1995, pp. 146-170. Also Technical Report CMU-CS-93-151, May 1993.

## 3.4 Daniel Jackson

My work focuses on fully automatic analyses of specifications and code. The approach taken by my group attempts to be at the same time principled and pragmatic. The models underlying our analyses are simple and well-understood mathematical models (such as the relational algebra and the typed lambda calculus), which allows to build powerful and predictable analysis engines. On the other hand, our tools and notations are motivated by practical case studies.Languages and analyses interest us only to the extent that they solve important industrial problems.

### 3.4.1 Specification analysis

Many large developments suffer enormous costs due to errors made early in requirements or design that are only discovered much later. By analyzing specifications automatically, many of these errors can be detected while they are still cheap to fix.

We have developed a tool called Nitpick that checks properties of software specifications, and simulates their execution. Hand-in-hand, we have developed a specification language called NP whose design attempts to optimize expressive power with the constraints of tractability. NP is roughly a subset of the Z specification language.

Many software artifacts can be specified in NP and checked by Nitpick. We have successfully analyzed a number of small but realistic examples, including the style mechanism of Microsoft Word, a mobile host routing protocol, and the directory/aliasing structure of the Macintosh Finder. Using a variety of reduction mechanisms, Nitpick can check huge numbers of cases; in one example, we checked $10^{23}$ transitions of an operation for the preservation of an invariant.

The fundamental premise of Nitpick/NP is that increased power of analysis can be gained by reducing the expressive power of the specification language and the comprehensiveness of the specification itself. Instead of following the traditional route of writing a full formal specification in a very expressive language such as Z, we construct a partial specification in a restricted language, NP, focusing on the particular aspects of behavior that are regarded as most likely to be the source of serious and costly errors. As a result, the cost of specification is dramatically reduced, and the analysis can be performed completely automatically. Measured purely in terms of the depth and breadth of the specification, our approach provides both less coverage and less power than a conventional approach. But like a surgical laser in comparison to a light bulb, the coherence of the approach overcomes the drawbacks in power and coverage, and results in greater benefits with much improved efficiency. We plan in the near future to apply Nitpick to larger case studies, in particular telephone call processing and railway switching. The NP language also seems to be well suited to expressing properties of architectural structure, and we plan to exploit this soon.

### 3.4.2 Code analysis

The crippling costs of maintenance are due largely to the inscrutability of large software systems. As soon as a system reaches more than a few thousand lines, it becomes difficult to extract even simple properties, such as how values pass from one global structure to another, or what aliasing patterns are in effect at various points in the code. And yet for almost all maintenance purposes, such properties must be known to the maintainer. For example, if the maintainer wants to eliminate the platform dependence of a system, it might be necessary to find all places in the code that access some handle to an external resource (a file descriptor, for example). Simple lexical tools such as UNIX grep, and even syntactic tools such as call-graph generators and cross-reference analyzers, cannot extract this kind of information.

We have built a tool called LACKWIT that can provide answers to such questions employing an efficient and uniform analysis. Essentially, LACKWIT identifies groups of values having the same representation, and tracks the propagation of these values through code and global data. This information is vital for many applications, such as the reverse-engineering of abstract data types, and can be exploited for many other purposes, such as identifying undesirable couplings between subsystems, and tracing the repercussions of a local modification.

LACKWIT differs from conventional program analyzers by using type inference as its underlying algorithm. This makes it highly efficient (the cost of a complete analysis is barely more than linear in the size of the program) while elegantly handling 'tricky' programming language features such as pointer aliasing and first-class functions. This kind of tool fills a gap by achieving the performance of a lexically-based analysis while still incorporating a large amount of semantic information. We have implemented an analyzer for C and are evaluating its results on industrial applications.

## References

[Jackson 95] D. Jackson. "Structuring Z Specifications with Views." ACM Transactions on Software Engineering and Methodology, October, 1995.

[Jackson and Jackson 96] D. Jackson and M. Jackson. "Problem Decomposition for Reuse." Software Engineering Journal, vol. 11, no. 1, pp. 11-30, January, 1996.

[Jackson et al 96] D. Jackson, S. Jha, and C. A. Damon. "Faster Checking of Software Specifications By Eliminating Isomorphs." Proceedings of Principles of Programming Languages, St. Petersburg Beach, FL, 1996.

[Jackson and Damon 96] D. Jackson and C. A. Damon. "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector." Proceedings of the International Symposium on Software Testing and Analysis, San Diego, CA, 1996.

[O'Callahan and Jackson 95] Robert O'Callahan and Daniel Jackson. "Detecting Shared Representations Using Type Inference." Technical Report CMU-CS-95-202, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1995.

## 4 Keywords

Architectural idioms (or styles)
Architectural design rules
Composability
Components
Concurrent and distributed systems
Connectors
Dataflow dependencies
Designer's associate
Formal methods
Formal specifications
Heterogeneous architectures
Interoperability
Interface specifications
Interface definition languages
Partial specification
Program slicing
Protocol specifications
Reverse engineering
Semantic match
Signature matching
Software architecture
Software analysis
Software frameworks
Specification Matching
Tractable representations