Capturing Design Expertise in Customized Software Architecture Design Environments

Robert T. Monroe
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract:

Software architecture is receiving increasing attention as a powerful way to deal with the complexity of large software systems. It has emerged as a distinct form of abstraction for software systems with its own set of design issues, vocabulary, and goals. Like designers in other disciplines, software architects can gain significant leverage by using powerful design environments and tools. Powerful design tools generally encapsulate a relatively small amount of *design expertise* that provides the important functionality of the tool within a relatively large support infrastructure. In this position paper I argue that in order to make the development of specialized architectural design tools practical it must be relatively easy and inexpensive to capture and make use of the desired design expertise. I then briefly describe an emerging approach for capturing this design expertise so that it can be used to incrementally configure architectural design environments.

1 Introduction

Software designers and developers have long realized the importance of powerful and appropriate abstractions for software systems. The *architectural* level of abstraction describes, at a relatively coarse level, the decomposition of a system into its major components, the mechanisms and rules by which those components interact, and the global properties of the system that emerge from the composition of its pieces. There is growing recognition in the software design community that one of the critical steps for the successful completion and fielding of a major software system is the creation of a well defined and documented architecture [5].

Given that architectural design is a critical step in the process of building a complex software system, one promising approach to improving the state of the practice of building software lies in providing software architects with powerful design environments and tools for specifying, analyzing, and reasoning about software architectures. The success of computer assisted design tools in other domains such as mechanical engineering, building architecture, and VLSI design argues that when design tools effectively capture the important aspects of design in a given domain they can offer useful analyses, significant reuse of common design elements, and even design guidance and evaluation. It is reasonable to believe that appropriate software architecture design tools can provide software architects with similar leverage.

In order for software architecture design tools to provide useful analyses and design guidance they must be capable of capturing, encoding, and making use of *architectural design expertise.* Architectural design expertise encompasses the concepts, models, rules and guidelines that skilled software architects use when specifying, constructing, or analyzing a software architecture. Examples of architectural design expertise include selecting an appropriate vocabulary of design elements, verifying the type-correctness of message communication paths between components, using rate-monotonic analysis to determine the schedulability of a real-time system, and determining whether the communications topology of a given architectural design provides sufficient redundancy and throughput to satisfy its requirements.

The traditional approach to building architectural design environments has been to hard-wire this expertise into the environment and its tools in an ad-hoc manner. This expertise is hard-wired not only in terms of the tool and environment implementations but also in terms of the selection of design concepts and models that the finished tool will support. Examples of design tools using this approach include UniCon [8], GenVoca [3], and the numerous commercial object-oriented design environments that support the Object Modeling Technique [7].

Because significant portions of design environments are developed on an ad-hoc basis, they are expensive, difficult, and time-consuming to build. An environment builder must come up not only with a design and implementation for the environment and the analyses it will support but also develop models for structuring, representing, and modifying the architectural designs produced with the environments. All of this work is in addition to the process of building any applicable supporting infrastructure. Moreover, once built these environments tend to be difficult to evolve, reuse, and adapt to new problems or domains unless they were originally and explicitly designed to support these modifications

Along with the Architecture-Based Languages and Environments (ABLE) group at Carnegie Mellon, I have been exploring a different approach to capturing and using architectural design expertise in software architecture design environments. This approach provides a customizable architectural design environment and a framework for capturing architectural design expertise using *architectural styles* and *design rules*. Once specified, these styles and design rules can be used to configure the generic architecture design environment, customizing the environment to support design done according to the styles and rules. Much of the infrastructure required for creating a style-specific architectural design environment – both the conceptual foundations and implementation artifacts – can then be reused when building a new environment. Further, the environment can be readily and inexpensively adapted to support new and different styles of design and enforce appropriate design rules.

The position that I am arguing then, is that the difficulty, cost, and time required to build useful architectural design environments can be greatly reduced by providing software architects with a standard framework for capturing architectural design expertise, along with a configurable architectural design environment that can be incrementally customized with this design expertise. In particular, this framework needs to extend beyond the notion of architectural style to include support for the specification and enforcement of style-independent design rules.

2 Capturing and using design expertise

In the remainder of this paper I will provide a brief overview of an approach to capturing design expertise and supporting the lightweight, incremental customization of architectural design environments to make use of this expertise. For the purposes of architectural design tool customization, the following complementary mechanisms are useful for capturing design expertise.

- **Architectural styles** capture broad properties, vocabulary, and configuration constraints that apply to all instances of a family of systems.
- **Design rules** capture more fine-grained guidelines, rules, and constraints that are most useful if expressed independent of an architectural style. Although styles can specify design rules that apply to all instances of the style, the scope of design rules can be less broad than a complete style. Design rules can be applied to individual design instances or even individual elements within a specific design.

2.1 Style-based capture of design expertise

The architectural style construct provides the conceptual framework for capturing design expertise and customizing the original Aesop architectural design environment [4]. Broadly, an architectural style specifies the vocabulary of design elements that can be used when designing in that style, composition constraints that specify topological and type constraints, and analyses that can be performed to evaluate emergent properties of a design. Table 1 provides a brief overview of the vocabulary, composition constraints, and analyses available for three different styles that Aesop supports. For a more detailed and formal treatment of architectural style see [1].

Style	**Design Vocabulary**	**Composition Constraints**	**Design Analyses**
Generic (null)	generic components and connectors	Any (component, component) pair can be attached with any connector.	• formal specifications consistency check (Wright language)
Pipe-and-Filter[a]	filter components and pipe connectors	(filter.output to filter.input) is the only valid attachment pair. The filters must be attached by a pipe.	• compilation, • throughput analysis • no cycles
Real-Time Producer/ Consumer[b]	process, device, and resource components. async-msg-pass and synchronous-msg-pass connectors	(process, process) and (device, process) pairs can only be attached with async-msg-pass connectors. (process, resource) pairs require sync. connectors.	• message-typechecking • schedulability analysis • processing rate calculation • repair heuristics

Table 1: Vocabulary, constraints and analyses for three selected architectural styles supported by Aesop.

a. For a detailed formal description of the Pipe-and-Filter style see [2].

b. For a full description of the Real-Time Producer/Consumer style see [6].

2.2 Limitations of a pure style-based approach

Our experience building styles and using Aesop indicates that styles are a reasonably effective way to capture broad design expertise and make use of that expertise to customize architectural design environments. It has also become apparent, however, that the effectiveness of styles for capturing fine-grained, context dependent design expertise is limited. Specifically, supporting only the style construct for capturing design expertise has the following three drawbacks:

1) **Styles are heavyweight, monolithic, and inflexible**. Because styles capture a lot of interrelated design expertise that has (hopefully) been shown to work appropriately together, modification of a style can have significant consequences beyond the point of modification. The tight intermingling of vocabulary, composition constraints, and analyses also makes it difficult to reuse pieces of a style definition in the creation of a new style.

2) **The range of expressibility of design expertise is limited**. A style specification defines invariant aspects of all systems built in the given style. As a result, if a particular piece of design expertise can not be readily expressed as a system invariant then it is difficult to make use of that piece of expertise in the style. Specifying design heuristics, for example, can be difficult or impossible.

3) **All vocabulary and composition rules have a scope of one complete style**. Styles provide little assistance for capturing or using design expertise that applies either across multiple styles, to only a context-dependent subset of the elements in a design, or that varies in applicability over time. As a result it is difficult to allow architects to use their judgement in relaxing or overriding rules as they find appropriate. Likewise, adding localized constraints to a design requires adapting the style itself.

2.3 Supporting incremental customization with design rules

To address these limitations I have been developing a lighter-weight framework for capturing design expertise that supports the incremental specification and enforcement of design rules within an environment but independent of a style. This new framework allows a designer to attach design rules to either design instances or (possibly singleton) sets of components and connectors within a specific design. By making the scoping of design rules more flexible and their specification much more modular than that of a complete style, an architect can add, remove, and disable design rules as appropriate for various stages and types of design.

Consider the example of an architect building a transaction processing system in a client-server style. He knows that the primary server will be able to handle no more than ten transactions per second. By specifying this as an independent design rule (and assuming that he has other tools specified to calculate the transaction rate) he can have the environment confirm that this constraint is being maintained as he creates and modifies the design. Enforcing this particular design rule over the complete style would probably be inappropriate because it would then require all designs created with that style to insure that the maximum transaction rate for the selected type of server was never exceeded. This particular rule is context-sensitive and varies in applicability from design to design.

A further example illustrates the claim that if design environments are to provide useful design guidance then there must be a role for design heuristics and suggestions that an architect can choose to ignore without invalidating the design. Consider the case of an architect working within an RPC-based client-server style where performance is sometimes, but not always, a significant concern. An appropriate design rule might be "first minimize cross-machine connections then minimize cross-process connections where possible." By specifying a design rule that watches the ratios of cross-machine and cross-process RPC connections to local procedure call connections (or, better yet, reusing a previously specified design rule) and associating that rule only with the performance critical pieces of the system, an architect can make simple modifications to the environment on the fly that address design concerns local to that instance. Because the design rule specification is independent of the style definition, the style definition does not need to be modified. As a result, other portions of the system remain unaffected by this additional constraint.

2.4 Implications for future research

Work is currently in progress on adapting Aesop to support the incremental specification and enforcement of style-independent design rules. Work to date has revealed the following four research issues as critical to the success of the undertaking:

1) **Developing appropriate design rule representations**. The representations used for specifying design rules must appear natural and flexible to the architects specifying the rules. Further, they must support the concise expression of the design rule and be extensible to deal with rules that were not originally envisioned by the environment creators. The initial approach used in Aesop has been to encapsulate these rules in declarative, predicate-based textual expressions.

2) **Providing unobtrusive and efficient enforcement of design rules**. Policies for detecting and dealing with violations of the design rules should be independent of the rule specifications. To make the design environment enjoyable to use, checking that design rules hold and flagging violations should not significantly slow the user's interaction with the environment. This requires an efficient framework for incrementally evaluating design rules.

3) **Scalability and reuse concerns**. A complex design environment might need to support hundreds of design rules concurrently, many of which will be reused across environments and designs. Highly modular design rule representations should both help the system scale in the number of design rules it can manage and support reuse of the rules.

4) **Managing interacting and possibly conflicting design rules**. Because a style is a self-contained entity, it should not contain conflicting design rules. Once the environment has been opened up to support the enforcement of arbitrary design rules, however, it is quite possible that multiple design rules will place conflicting requirements on a design. One approach to dealing with this problem is to associate a priority with each rule and, in the case of a tie, request that the architect decide which rule(s) to obey and which to relax.

3 References

[1] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of SIGSOFT '93: Foundations of Software Engineering,* Software Engineering Notes 18(5), pages 9-20. ACM Press, December 1993.

[2] Robert Allen and David Garlan, Towards Formalized Software Architecture, Carnegie Mellon University Technical Report CMU-CS-TR-92-163, July 1992.

[3] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, Marty Sirkin. The GenVoca Model of Software System Generators. *IEEE Software*, September 1994, pp. 89 - 94.

[4] David Garlan, Robert Allen, and John Ockerbloom. Exploiting Style in Architectural Design Environments, in *Proc. of SIGSOFT '94 Symposium on the Foundations of Software Engineering*, Dec. 1994.

[5] *Proceedings of the First International Workshop on Architectures for Software Systems*, Edited by David Garlan. April, 1995.

[6] Kevin Jeffay. The Real-Time Producer/Consumer Paradigm: A Paradigm for the Construction of Efficient, Predictable Real-Time Systems, in *Proc. of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pp 796-804, Indianapolis, IN, Feb. 1993, ACM Press.

[7] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[8] Mary Shaw, Robert Deline, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, April 1995.