Software Architecture Critics in Argo

Jason E. Robbins **David M. Hilbert** **David F. Redmiles**

Dept. of Information and Computer Science
University of California, Irvine
Irvine, California, 92697-3425 USA
+1-714-824-7308
{jrobbins,dhilbert,redmiles}@ics.uci.edu

ABSTRACT
Software architectures are high-level design representations of software systems that focus on composition of software components and how those components interact. Software architectures abstract the details of implementation and allow the designer to focus on essential design decisions. Regardless of notation, designers are faced with the task of making good design decisions, which demands a wide range of knowledge of the problem and solution domains. Argo is a software architecture design environment that supports designers by addressing several cognitive challenges of design. In this paper we describe how Argo supports decision making by automatically supplying knowledge that is timely and relevant to decisions at hand.

Keywords
Domain-oriented design environments, software architecture, human cognitive needs, design critics

INTRODUCTION

Software architecture is one promising approach to the development of large software systems [6, 11, 12]. Software architectures are high-level design representations of software systems that focus on composition of software components and how those components interact. Software architects build systems by choosing and composing software components. Ideally, off-the-shelf components are reused, but in practice some components must be customized or developed from scratch. This component-based approach to development relies on notations that abstract the details of implementation and allow the designer to focus on essential design decisions.

An essential activity in designing software systems is decision making. The resulting design must satisfy the requirements and not violate constraints imposed by the problem domain and implementation technologies. In addition to hard constraints on the design, there are numerous soft constraints, or rules of thumb, that address qualities of the system that are desirable but not strictly required. Often there are several desirable qualities that conflict with each other or that cannot be precisely measured.

In the domain of software architectures, the designer must make myriad decisions, including the choice of software components, configuration of individual components, communication relationships between components, and allocation of available resources to components. Requirements are typically stated in terms of needed functionality, limitations on resource use, and non-functional requirements such as extensibility, portability, or scalability.

Software architects make design choices based on knowledge of available software components and resources and their characteristics. For example, in architecting a web page editing tool, one spell checking component may run on multiple platforms but be difficult to extend to handle HTML syntax, while another might be fast and flexible but require a run-time licensing fee, and developing a new spelling component demands time, budget, and specific technical skills.

PROBLEM

Typically no one designer has all the knowledge needed to make a complete design. Instead, most complex systems are designed by teams of stakeholders with each stakeholder providing some of the needed knowledge. Even experienced designers need knowledge support in complex domains or when working with new design materials. The “thin spread of application domain knowledge’’ has been identified as a general problem in software development [1].

The knowledge that designers use to make design decisions is diverse, heuristic, and tacit. Diverse knowledge is needed to address the diversity of design issues in a complex system. Design knowledge is often heuristic because qualities are difficult to measure or the relationship between design constructs and qualities is unclear. Because it is diverse and heuristic, designers cannot articulate or catalog all the knowledge they have, and it is impractical for tools to assume they can supply all needed knowledge.

In addition to the need for knowledge, designers also have difficulty applying the knowledge that is available to them. The cognitive theory of *reflection-in-action* [9, 10] states that designers can best evaluate their designs while they are engaged in making design decisions, not after. Availability bias arises when multiple pieces of information influence a decision, but only some of them are readily available, often resulting in decisions based on incorrect assumptions that must be reworked later [13].

Two other cognitive challenges of design are described by the theory of *opportunistic design* and the theory of *comprehension and problem solving*. *Opportunistic design* [4, 14] observes that designers do not follow prescribed design processes, instead they choose what to do next as they work through the design, based on perceived degree of difficulty and priority. *Comprehension and problem solving* [5, 7] addresses the way designers understand systems when engaged in making design decisions. Designers often use multiple mental models of the system, each one of which addresses a subset of design issues. Elsewhere [8] we describe features of Argo that address the cognitive challenges of design raised by these theories. In the following sections we focus on designers' need for knowledge and how Argo's critiquing infrastructure delivers that knowledge.

APPROACH

Traditional approaches to software analysis follow the *authoritative assumption*: they support architectural evaluation by proving the presence or absence of well defined properties. This allows them to give definitive feedback to the architect, but limits their application to late in the design process after the architect has formalized substantial parts of the architecture.

Critics are active agents that support decision-making by continuously and pessimistically analyzing partial architectures. Each critic checks for the presence of certain conditions in the partial architecture. Due to their continuous and pessimistic nature, however, care must be taken to ensure that critics do not distract the architect by providing an overwhelming volume of feedback. *Criticism control mechanisms* are used to control the execution of critics and manage their feedback, so as to inform the architect without distracting from the design task at hand. Critics are embedded in a design environment where they have access to the architecture as it is being modified and to a model of the design process as it is being enacted. Figure 1 shows an overview of Argo. Figure 2 shows a screenshot of Argo modeling an example architecture.

The critic-based approach makes what we call the *informative assumption*: architects are capable of making design decisions, and analysis is used to support architects by informing them of potential problems and pending decisions. Critics are written to pessimistically detect potential problems. They need not go so far as to prove the presence of problems; in fact, formal proofs are often not possible, or meaningful, on partial architectures. This approach avoids the need to assume that critics have complete knowledge, and facilitates incremental development and improvement of critics.

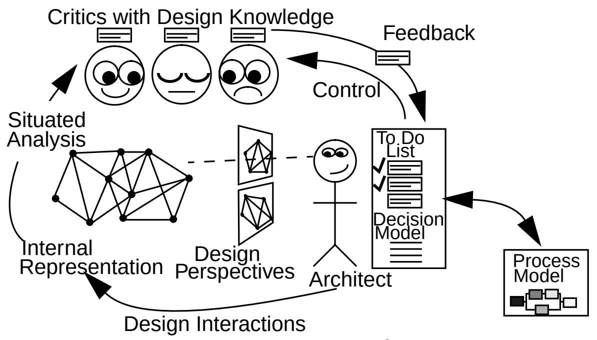
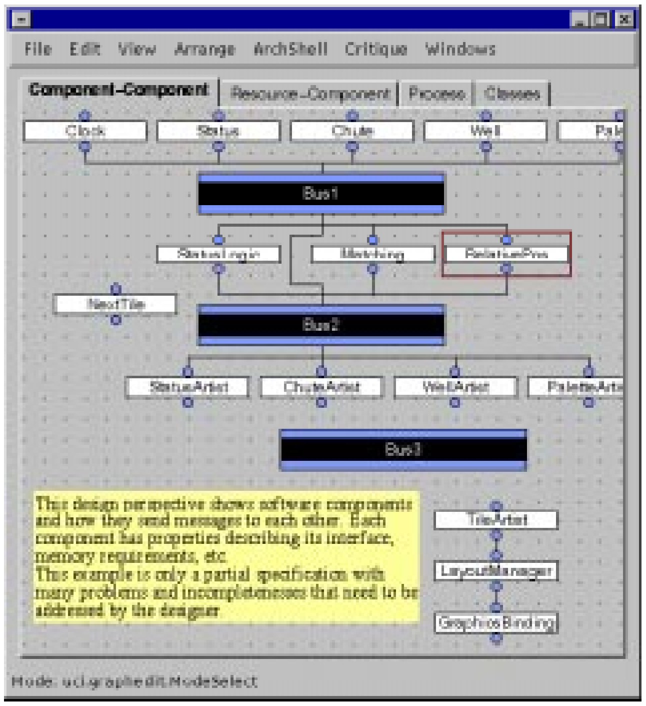


Figure 1. Design Environment Facilities of Argo



This design perspective shows software components and how they send messages to each other. Each component has properties describing its interface, memory requirements, etc.
This example is only a partial specification with many problems and incompletenesses that need to be addressed by the designer.

```
Mode: uci.graphedit.ModeSelect
```

Figure 2. A partially specified architectural model of a system

FEATURES OF ARGO

Critics

Critics can deliver knowledge to architects about the implications of, or alternatives to, a design decision. In the vast majority of cases, critics simply advise the architect of potential errors or areas needing improvement in the architecture; only the most severe errors are prevented outright, thus allowing the architect to work through invalid intermediate states of the architecture. Architects need not know that any particular type of feedback is available or ask for it explicitly. Instead, they simply receive feedback as they manipulate the architecture. Feedback is often most valuable when it addresses issues that the architect had previously overlooked and might never seek to investigate without prompting.

Each critic performs its analysis independently of others, checking one predicate, and delivering one piece of design feedback. Critics encapsulate domain knowledge of a variety of types. Correctness critics detect syntactic and semantic flaws. Completeness critics remind the architect of incomplete design tasks. Consistency critics point out contradictions within the design. Optimization critics suggest better values for design parameters. Alternative critics present the architect with alternatives to a given design decision. Evolvability critics consider issues, such as modularization, that affect the effort needed to change the design over time. Presentation critics look for awkward use of notation that reduces readability. Tool critics inform the architect of other available design tools at the times when those tools are useful. Experiential critics provide reminders of past experiences with similar designs or design elements. Organizational critics express the interests of other stakeholders in the development organization. These types serve to aggregate critics so that they may be understood and controlled as groups. Some critics may be of multiple types, and new types may be defined, as appropriate for a given application domain. Table 1 shows some example architecture critics.

Criticism Control Mechanisms

Formalizing the analyses and rules of thumb used by practicing software architects could produce hundreds or thousands of critics. To provide the architect with usable information, a subset of these critics must be selected for execution at any given time. Critics must be controlled so as to make efficient use of machine resources, but our primary focus is on effective interaction with the architect.

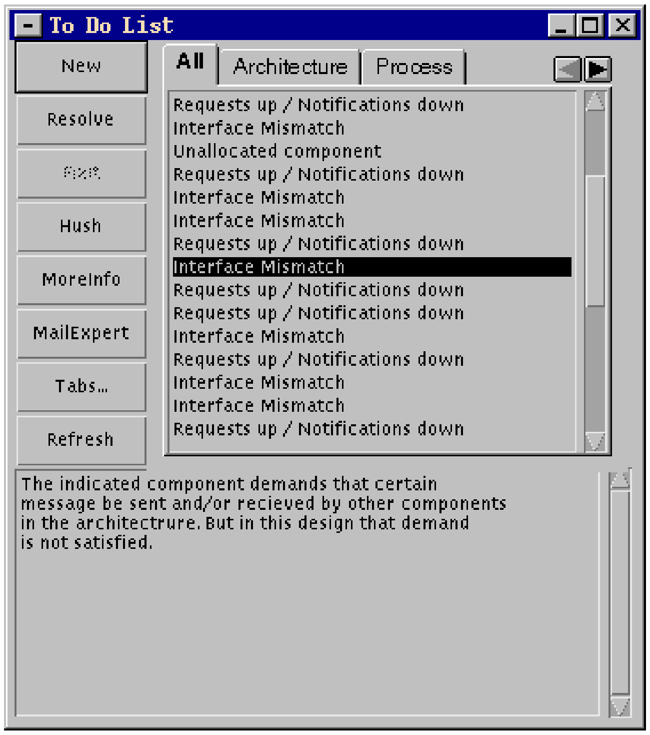
Criticism control mechanisms are predicates used to limit execution of critics to when they are relevant and timely to design decisions being considered by the architect. For example, critics related to "fine tuning" an architecture should not be active when the architect is working on its rough organization. Attributes on each critic identify what type of design decision it supports. Criticism control mechanisms check those attributes against the design goals and process model. Computing relevance and timeliness separately from critic predicates allows critics to focus entirely on identifying problematic conditions in the product (i.e., the partial architecture) while leaving cognitive design process issues to the criticism control mechanisms. This separation of concerns also makes it possible to add value to existing critics by defining new control mechanisms.

Feedback Management

Once critics generate design feedback, it must be presented to the architect in a usable form without distracting the architect. In Argo, the "to do" list user interface presents feedback to the architect (Figure 3).When the architect selects a pending feedback item from the upper pane, the associated (or "offending") architectural elements are highlighted in all design perspectives and details about the open design issue and possible resolutions are displayed in the lower pane. The designer may also follow links to background domain knowledge relevant to the issue at hand, or send email to the person who authored the critic. Together these pieces of information provide a design context that the designer can use to resolve the issue at hand. Providing contact information for relevant stakeholders helps to situate the problem and possible solutions in the context of the development organization.

RELATED WORK

Our focus on the cognitive needs of architects stems from the work of Fischer and colleagues [2]. In applying design environments to software architecture [8], we extended previous design environment facilities to support cognitive needs identified in the cognitive theories of *reflection-in-action* (via critics), *opportunistic design* (via a process model and "to do" list"), and *comprehension and problem solving* (via multiple-coordinated views [6, 12]).



Requests up / Notifications down
Interface Mismatch
Unallocated component
Requests up / Notifications down
Interface Mismatch
Interface Mismatch
Requests up / Notifications down
Interface Mismatch
Requests up / Notifications down
Requests up / Notifications down
Interface Mismatch
Requests up / Notifications down
Interface Mismatch
Interface Mismatch
Requests up / Notifications down

The indicated component demands that certain
message be sent and/or recieved by other components
in the architectrue. But in this design that demand
is not satisfied.

Figure 3. Argo's to do list indicating several open design issues

Table 1. Examples of Architecture Critics

Name of Critic	Explanation
Interface Mismatch	This component needs the certain messages be sent or received, but they are not present.
Direct Connection	Violation of C2 style guidelines. Consider using a message bus to allow new components later.
Component Choice	Here are other components that could "fit" in place of what you have: <<list of components>>.
Too Much Memory	Calculated memory requirements exceed stated goals
Too Many Components	There are too many components at the same level of decomposition to be easily understood.
Hard Combination to Test	If you need to use these components together, please make arrangements with the testing manager.
Generator Limitation	The default code generator can not make full use of this component.
Not Enough Reusable Components	The fraction of components marked as being reusable is below your stated goals.

Aesop [11] is a tool that generates style-specific software architecture design environments from a set of formal style descriptions. Aesop primarily addresses requirements of architecture representation, manipulation, visualization, and analysis, without providing explicit support for evolutionary design or the architect's cognitive needs.

STATUS AND FUTURE WORK

It is our goal to develop and distribute a reusable design environment infrastructure that others may use, extend, and integrate with their research to better support architectural evolution and architects' cognitive needs. To date we have produced two prototypes. The current version is implemented in Java and is available with documentation via http://www.ics.uci.edu/pub/arch.

In future work we will continue the themes of our current research and increase the functionality of our prototype. The identification of new theories of cognitive challenges may result in new features. Also, Argo provides the potential to measure the impact that various kinds of knowledge have on designers' decisions. We plan to explore issues of design rational: critics may serve as a foil[1] that encourages designers to make rational more explicit, and the rational for previous decisions is an important part of the design context of current issues.

ACKNOWLEDGEMENTS

Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and F30602-94-C-0218, and by the National Science Foundation under Contract Number CCR-9624846. Additional support is provided by Rockwell International. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Approved for Public Release — Distribution Unlimited.

1. In acting terminology, a foil is a minor character who serves to highlight and clarify a major character, usually through dialog.

REFERENCES

1. Curtis, W., Krasner, H., and Iscoe, N. A field study of the software design process for large systems. *Comm. ACM*. 1988. vol. 31, no. 11. pp. 1268-1287.
2. Fischer, G. Domain-oriented design environments. *Proc. 7th Knowledge-Based Software Engineering Conference*. pp. 204-213.
3. Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., and Sumner, T. Embedding computer-based critics in the contexts of design. *INTERCHI'93*. April 1993. pp.157-164.
4. Guindon, R., Krasner, H., and Curtis, W. Breakdown and processes during early activities of software design by professionals. In: Olson, G. M. and Sheppard S., eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corporation. 1987. pp. 65-82.
5. Kintsch, W. and Greeno, J. G. Understanding and solving word arithmetic problems. *Psychological Review.* 1985. vol. 92. pp. 109-129.
6. Kruchten, P. B. The 4+1 view model of architecture. *IEEE Software.* Nov. 1995. pp. 42-50.
7. Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology.* 1987. vol. 19. pp. 295-341.
8. Robbins, J. E., Hilbert, D. M., Redmiles, D. F. Extending design environments to software architecture design. *Proc. 11th Knowledge-Based Software Engineering Conference'96*. pp. 63-72.
9. Schoen, D. *The Reflective Practitioner: How Professionals Think in Action*. 1983. New York: Basic Books.
10. Schoen, D. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems*. 1992. vol. 5, no. 1. pp. 3-14.
11. Shaw, M., Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
12. Soni, D., Nord, R., Hofmeister, C. Software Architecture in Industrial Applications. *Proc. Inter. Conf. on Software Engineering 17*. 1995. pp. 196-207.
13. Stacy, W. and MacMillian, J. Cognitive Bias in Software Engineering. *Comm. ACM*. June 1995. pp. 57-63.
14. Visser, W. More or Less Following a Plan During Design: Opportunistic Deviations in Specification. *Int. J. Man-Machine Studies*. 1990. pp. 247-278.