Using Critics to Support Software Architects

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

{jrobbins,dhilbert,redmiles}@ics.uci.edu

Technical Report UCI-ICS-97-18
Department of Information and Computer Science
University of California, Irvine
Irvine, California, 92697-3425
April 1997

ABSTRACT Software architectures evolve as the result of numerous, interrelated design decisions. Existing approaches to analysis, however, tend to provide feedback only after numerous design decisions have been made. As a result, they do not directly support the evolutionary nature of the architecture design process or the software architect's decision-making process. In this paper we present an approach to architectural analysis stemming from previous work in domain oriented design environments that is based on critics and criticism control mechanisms. This approach more closely supports evolution and the needs of architects by providing feedback as individual design decisions are being considered. We discuss the theoretical motivations for the critic-based approach, the implementation and management of critics, support for diverse and extensible groups of critics, and the combined use of critics and existing analysis techniques.

KEYWORDS Software architectures, architectural analysis, critics, design environments, human-computer interaction, evolutionary design

Using Critics to Support Software Architects

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

Information & Computer Science
University of California, Irvine
Irvine, CA, USA
(714) 824-7308, (714) 824-3100, (714) 824-3823
{jrobbins,dhilbert,redmiles}@ics.uci.edu

ABSTRACT

Software architectures evolve as the result of numerous, interrelated design decisions. Existing approaches to analysis, however, tend to provide feedback only after numerous design decisions have been made. As a result, they do not directly support the evolutionary nature of the architecture design process or the software architect's decision-making process. In this paper we present an approach to architectural analysis stemming from previous work in domain oriented design environments that is based on critics and criticism control mechanisms. This approach more closely supports evolution and the needs of architects by providing feedback as individual design decisions are being considered. We discuss the theoretical motivations for the critic-based approach, the implementation and management of critics, support for diverse and extensible groups of critics, and the combined use of critics and existing analysis techniques.

Keywords

Software architectures, architectural analysis, critics, design environments, human-computer interaction, evolutionary design

INTRODUCTION

Software architectures, like the systems they model, are products of evolutionary design processes. If architecture is to fulfill its potential in supporting software design, it must be accompanied by analysis techniques that take into consideration the evolutionary nature of the architecture design process as well as the cognitive needs of software architects.

Existing approaches to architectural analysis are coarse-grained and discrete. Design decisions are entered into a formal representation. That formal representation is fed as input to analysis tools which produce output regarding properties of the representation. Finally, architects interpret the output, relate it back to design decisions embodied in the representation, and prepare the design for another iteration. In sum, existing approaches require the architect to suspend the evolution of the architecture by creating a snapshot for analysis and, consequently, to suspend or delay the decision-making process by clustering modifications between evaluation opportunities. This design process is coarse-grained, operating on whole architectures as units. The cognitive process is correspondingly coarse-grained, dealing with clusters instead of individual decisions.

In contrast to this coarse-grained, discrete approach, we propose a fine-grained, concurrent approach. Namely, we advocate the use of critics to perform analysis on partial architectural representations *while* architects are considering individual design decisions and modifying the architecture. Analysis is concurrent with decision-making so that architects are not forced to suspend the architecture's evolution or cluster their decisions in preparation for analysis. Feedback from critics can be used by architects while they are considering design decisions. Furthermore, critic feedback is directly linked to elements of the architecture thereby assisting architects in applying the feedback in revising the design. We believe this approach more directly supports the evolutionary nature of the architecture design process and the cognitive needs of software architects.

The critic-based approach to analysis extends on-going research by Fisher and colleagues as discussed in the related work section. Our work in applying the approach to software architecture has led us to investigate a variety of new theoretical and implementation challenges. Specifically, we discuss four challenges faced by architectural analysis techniques intended to support software architects: (1) the evolutionary nature of the architecture design process, (2) the cognitive needs of architects, (3) support for diverse analyses, and (4) trade-offs between authoritative versus informative approaches. We then present implementation issues of the critic-based approach to architectural analysis. An extensive discussion section describes in detail how our approach addresses the problems outlined above, and how our approach is evaluated.

ARCHITECTURE DESIGN PROBLEMS

The Evolutionary Nature of Architecture Design

Software architectures are evolutionary artifacts. They are evolutionary in that they are constructed incrementally as the result of many interrelated design decisions made over extended periods of time. We visualize design as a process in which a path is traced through a space of branching design alternatives [27]. A particular software architecture can be thought of as a product of one of the possible paths through

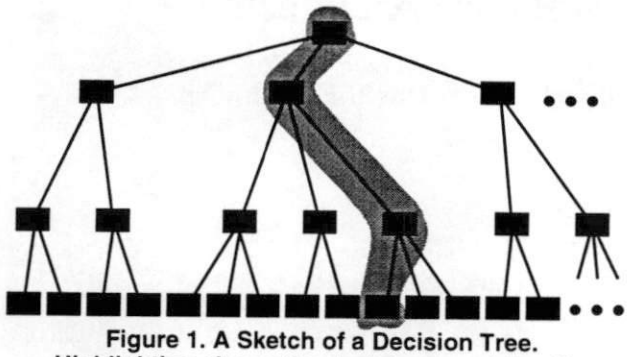

Figure 1. A Sketch of a Decision Tree. Highlighting shows the decisions contained in one complete design.

this space (Figure 1). Choices at any point can critically affect alternatives available later, and every decision has the potential of requiring earlier decisions to be reconsidered.

Analysis techniques that require architects to make numerous design decisions before feedback is provided do not directly support the evolutionary nature of the architecture design process. In terms of our sketch in Figure 1, such analyses evaluate the products of relatively complete paths through design space, without providing much guidance at individual branching points. As a result, substantial effort may be wasted by building upon poor decisions before feedback is available to indicate the existence of problems. Furthermore, when analysis is performed only after extended design episodes, it may be difficult to identify where exactly in the decision path the architect initially went wrong. When substantial effort is required to formalize details of an architecture before it can be analyzed, fewer design alternatives will be explored, thus reducing confidence in the design.

The Cognitive Needs of Architects

Design is a cognitively challenging activity. Much research in cognitive science has focused on the cognitive needs of designers in other fields, including traditional architecture. Since software architects are clearly engaged in a complex design task, approaches to architectural analysis should take into account issues raised by cognitive scientists.

Schoen's theory of *reflection-in-action* [31, 32] observes that designers of complex systems do not conceive a design fully-formed. Instead, they must construct a partial design, evaluate, reflect on, and revise it, until they are ready to extend it further. Guindon, Krasner, and Curtis noted the same effect as part of one study of software developers [15]. Calling it "serendipitous design," they noted that as the developers worked hands-on with the design, their mental model of the problem situation, and hence their design, improved.

The cognitive theory of *opportunistic design* describes how designers deviate from plans, even their own plans, in order to minimize the cognitive cost of context switches between design tasks [27, 36, 37, 40]. For example, if a decision raises design issues that require a deviation from the current process, the architect must either deviate, or mentally defer those issues in order to continue with the current process. While deviations from design plans may be desirable from a cognitive perspective, they may lead designers into a variety of difficulties as discussed in the Guindon, Krasner, and Curtis study [15].

Finally, the theory of *comprehension and problem-solving* observes that designers benefit from seeing their designs from different design perspectives [8, 16, 23]. The use of multiple perspectives divides the complexity of the architecture and separates concerns [17, 21, 35]. The availability of multiple perspectives also increases the chance that an architect will see a simple mapping between one of them and his or her mental model of the problem being addressed [24]. Coordination among design perspectives means that design elements and relationships presented in multiple perspectives may be viewed and manipulated in any of those perspectives. Overlap among perspectives provides shared context and allows the architect to apply knowledge of one perspective to another, thereby aiding understanding of new perspectives, and new design issues [29, 30].

Support for Diverse Analyses

Diverse analyses are required to support architects in addressing diverse design issues, such as performance, security, fault-tolerance, and extensibility. The need for diversity in analysis is further driven by the diversity in project stakeholders and the potentially conflicting opinions of experts in the software architecture field itself [12, 21].

Conflict will naturally arise in architecture design, and analysis techniques should be capable of accommodating it. Accommodating conflict in analysis yields more complete support, whereas, forbidding conflict essentially prevents architects from viewing a design issue from more than one viewpoint.

Consider, for example, architectural styles [33, 38]. Styles define the vocabulary of an architecture and a set of rules that determine if an architecture is well formed. Architectural styles provide design guidance by suggesting constraints on design decisions. A given architecture may nearly satisfy the rules of several diverse styles simultaneously, and analytical feedback related to each of those styles might be useful, even if conflicts arise.

Authoritative Versus Informative Approaches

Existing software analysis techniques are extremely powerful for detecting well-defined properties of completed systems, such as memory utilization, performance, and the possibility of deadlock. These approaches adhere to what we call 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 may limit their application to late in the design process, after the architect has committed substantial effort building upon unanalyzed decisions.

Such approaches also tend to use an interaction model that places a substantial cognitive burden on architects. For example, architects are usually required to know of the availability of analysis tools, recognize their relevance to particular design decisions, explicitly invoke them, and relate their out-

put back to the architecture. This model of interaction draws the architect's attention away from immediate design goals and toward the steps required to get analytical feedback.

We propose an alternative approach to architectural analysis based on what we call the *informative assumption*: architects are ultimately responsible for making design decisions, and analysis is used to support architects by informing them of potential problems and pending decisions. Analyses need not go so far as to prove the presence of problems, since formal proofs are often not possible, or even meaningful, on partial architectures. However, such proofs can be produced under our approach if it is possible to produce them at all.

Because formal proofs are not required, heuristics can be used to identify potential problems. This is valuable because heuristic analyses can identify problems involving design details that may not be explicitly represented in the architecture, either because the representation abstracts away relevant information, or because the architecture is only partially specified. For example, if limited memory is a design constraint, but the mapping of modules to hosts is not explicitly represented yet, then estimating host memory requirements cannot be done authoritatively. However, certain features of the architecture (for instance, several concurrently executing, communicating modules) may suggest that memory usage is an issue the architect should investigate before going further.

THE CRITIC-BASED APPROACH

We address the problems raised above through the use of critics and criticism control mechanisms [11]. *Critics* are agents that support decision-making by continuously and pessimistically analyzing partial architectures and delivering design feedback. *Criticism control mechanisms* are used to control the execution of critics and manage their feedback. The goal is to inform the architect without distracting from the design task at hand.

The critic-based approach supports evolution and the needs of software architects by providing continuous feedback as design decisions are made. Analyses can be numerous and diverse and are implicitly invoked without requiring overt knowledge or action on the part of the architect. In addition to analysis of well-defined formal properties, critics deliver design guidance based on rules of thumb, style guidelines, empirical data, and other heuristics.

The critic-based approach is neither a replacement for, nor incompatible with, authoritative analysis techniques. Critics are simply an alternative way of packaging and using analyses, and can be used in conjunction with existing tools, as will be discussed later. Critics and criticism control mechanisms operate within the context of a *design environment* [8, 10].

The Design Environment

Figure 2 presents selected facilities of Argo, our critic-based design environment for software architecture. The architect uses multiple, coordinated design perspectives (Figure 3) to view and manipulate Argo's internal representation of the architecture, which is stored as an annotated, connected

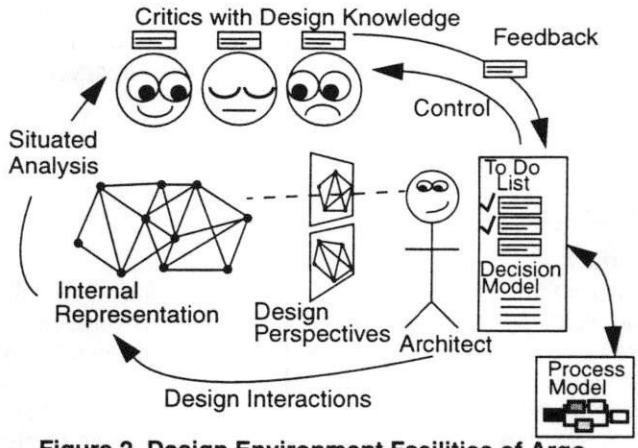

Figure 2. Design Environment Facilities of Argo

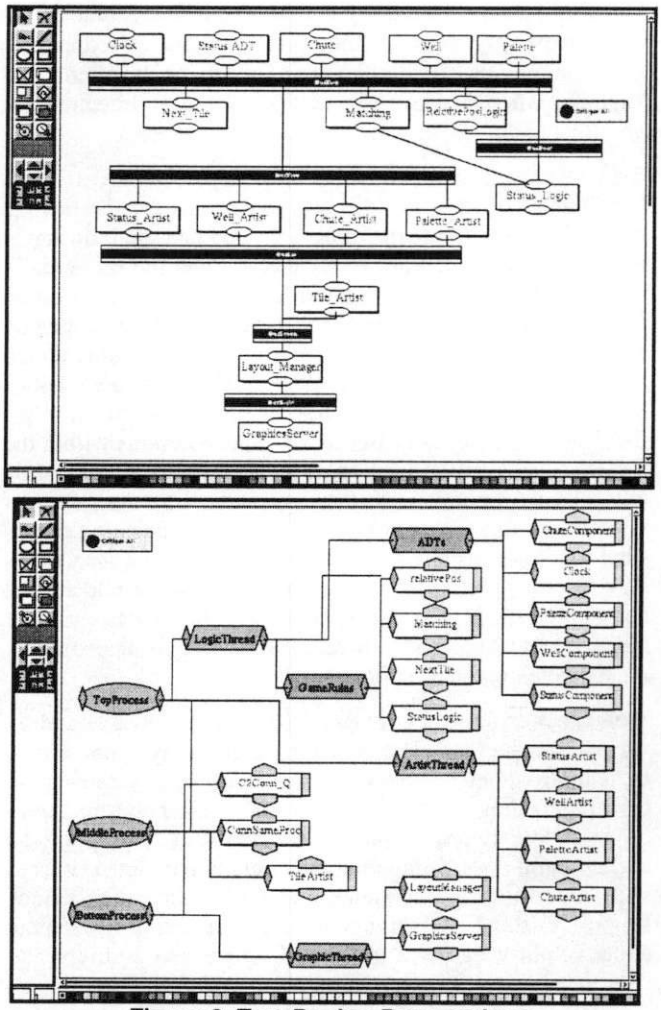

Figure 3. Two Design Perspectives: Conceptual and Execution

graph. The various perspectives are projections, or subgraphs, of the internal representation such that each perspective presents only architectural elements relevant to a limited set of related issues.

Critics monitor the partial architecture as it is manipulated, placing their feedback in the architect's "to do" list (Figure 4). Argo's process model serves the architect as a resource in carrying out an architecture design process, while the decision model lists issues from the process model that the architect is currently considering. Criticism control mechanisms use that decision model to ensure the timeliness of feedback.

Critics

Critics deliver knowledge to inform architects of the implications of, or alternatives to, design decisions. 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 aspects of the architecture the architect had not specifically chosen to analyze.

Critics encapsulate domain knowledge, such as well-formedness of the design, hard constraints on the design, rules of thumb about what makes a good design, industry or organizational guidelines or style rules, and the (potentially conflicting) opinions of domain experts. We can define a variety of types of critics, each type delivering a specific kind of knowledge. Correctness critics detect syntactic and semantic flaws in the partial design. Completeness critics detect when a design task has been started but not yet finished. Consistency critics detect contradictions within the design. Presentation critics detect awkward use of the notation. Alternative critics remind the designer of alternatives to a given design decision. Optimization critics suggest better values for design parameters. 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 need to be defined, as appropriate, for a given application domain.

We expect critics to be invented for various reasons and by various stakeholders. Practicing architects may define critics to capture their experience in building systems and distribute those critics to other architects in their organization. Researchers may define critics to support an architectural style. Component vendors may define critics to add value to the components that they sell, and to reduce support costs. Critics may be implemented to speculate about implications of a given decision based on empirical data that indicates correlations. Also, existing literature on architectural styles and system design provides advice that can be made active via critics.

Some examples of architecture critics are given in Table 1. These critics perform fairly simple analyses that are meaningful for partial architectures. As the architecture becomes more fully specified, critics may also make use of external analysis tools for in-depth analyses (see "DISCUSSION AND EVALUATION").

Criticism control mechanisms

Each critic performs its analysis independently of others, checking one predicate, and delivering one piece of design feedback. Criticism control mechanisms determine which critics are relevant and timely to design decisions being considered by the architect. Critics are implicitly made runnable when the control mechanisms determine that they are relevant and timely. 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 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.

The "To Do" List

In order for the critic-based approach to scale up, design feedback must be managed so as not to overwhelm or distract the architect. The "to do" list user interface presents design feedback to the architect (Figure 4). Critics post "to do" items as a result of their analyses. The process model posts "to do" items to remind the architect to finish tasks that are in progress. The architect may also post "to do" items as notes or reminders to return to deferred design explorations. Architects may address issues in any order they choose, and the list of items may be filtered and sorted based on various attributes.

Each "to do" item refers back to the architecture, critics, goals, and process model. When the architect selects an item from the upper pane of the window in Figure 4, the associated (or "offending") architectural elements are highlighted in all design perspectives and the lower pane displays details about the open design issue and possible resolutions. Once an item is selected, the architect may manipulate the critic that produced that item, send email to the expert who authored that critic, or follow hyperlinks to more information.

The "to do" items are a potentially rich source of data for design rationale. Items are placed on the list to identify open issues, and removed from the list when those issues are

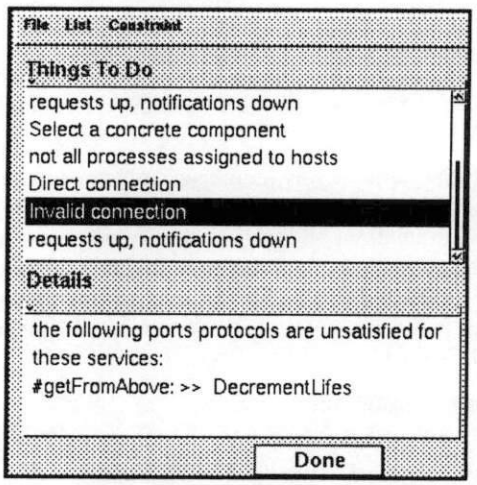

Figure 4. The architect's "to do" list

Name of Critic	Critic Type	Decision Category	Explanation
Invalid Connection	Correctness	Connecting	Mandatory message signatures not satisfied by adjacent components in the conceptual architecture
One Up One Down	Correctness	Connecting	Violation of C2 configuration rules
Simpler Comp. Avail.	Alternative	Choosing	A "smaller" component will "fit" in place of what you have
Too Much Memory	Consistency	Resources	Calculated memory requirements exceed stated goals
Need More Reuse	Consistency	Choosing	Percentage of reusable components is below stated goals
OS Incompatibility	Consistency	Choosing	Components have conflicting environmental requirements

Table 1. Selected Argo architectural critics

resolved. When an item is resolved, it can be appended to a log for later use as rationale. A description of the resolution can be entered by the architect or, in some cases, generated automatically. We will explore this possibility further in future research.

IMPLEMENTATION ISSUES

This section discusses the implementation of the Argo critic-based design environment for software architecture. Our discussion is based on two prototypes. The initial version, coded in Smalltalk, was demonstrated at ICSE-17. The current version is implemented in Java. First we discuss the core elements of our approach: critics and criticism control mechanisms. We then describe Argo's own architecture and the representation of architectures being designed, paying attention to issues that affect critic authoring. Finally, we discuss how the critic-based approach can be integrated with external analysis tools.

Critics

In Argo, a critic is implemented as a combination of an analysis predicate, attributes for determining relevance, and a "to do" list item to be given as design feedback. The stored "to do" list item contains a headline, a description of the issue at hand, contact information for the critic's author, and a hyperlink to more information.

Criticism control mechanisms select critics for execution. During execution a critic evaluates its analysis predicate and, if appropriate, constructs a "to do" list item and posts it. We encode critics as programming language predicates; deciding on what languages are best for expressing critics is a topic for future research. Table 2 presents a connection checking critic in detail. Critics may also place annotations on architectural elements as a side effect of critiquing. For example, once the messages flowing across a connector have been computed, that information can be cached in the connector.

Criticism Control Mechanisms

Criticism control mechanisms ensure relevance and timeliness by using explicit models of the architect's goals and process. Argo uses a combination of different criticism control mechanisms to determine whether each critic should be runnable. Predefined control mechanisms check a critic's type against a table of runnable types, check a critic's decision category against the decision model, or check a critic's hushed state.

Attribute	Value
Name	Invalid Connection
Type	Correctness
Decision Category	Connecting
Smalltalk Predicate	`[:comp | | invalidServices | invalidServices := comp inputs , comp outputs select:[:s | s isSatisfied not]. invalidServices isEmpty not. ]`
Description	"The following port protocols are unsatisfied for these services:" <<a list of ports and services>>
More Info	http://www.ics.uci.edu/pub/arch/
Expert	jrobbins@ics.uci.edu

Table 2. Details of the Invalid Connection Critic

Argo's user interface allows groups of critics to be made runnable or unrunnable by type. Architects may also "hush" individual critics — rendering them temporarily unrunnable — if their feedback is felt to be inappropriate or too intrusive. If the architect does not understand a particular critic, or believes it to be incorrect, Argo provides a way to send structured email to the maintainer of that critic. These control mechanisms were chosen to provide a range of semantic depth and user effort, although new control mechanisms may be implemented and combined with these.

Argo models the design process as a task network, where each task is focused on design decisions of a certain type. The architect marks each task as being done, in progress, or future. This marking determines which decisions should be listed in the decision model as decisions the architect is currently considering. With varying degrees of effort, Argo's process model, decision model, and "to do" list can be integrated with full-scale process tools, such as Endeavors [5].

Argo provides a critic run-time system that selects runnable critics and executes them in a separate thread of control. Critics may also be triggered by specific architecture manipulations. Another thread of control periodically re-examines each "to do" list item and removes items that are no longer applicable.

Design Environment Architecture

Figure 5 shows a screen shot of Argo modeling its own architecture in the C2 style [38]. The topmost row of

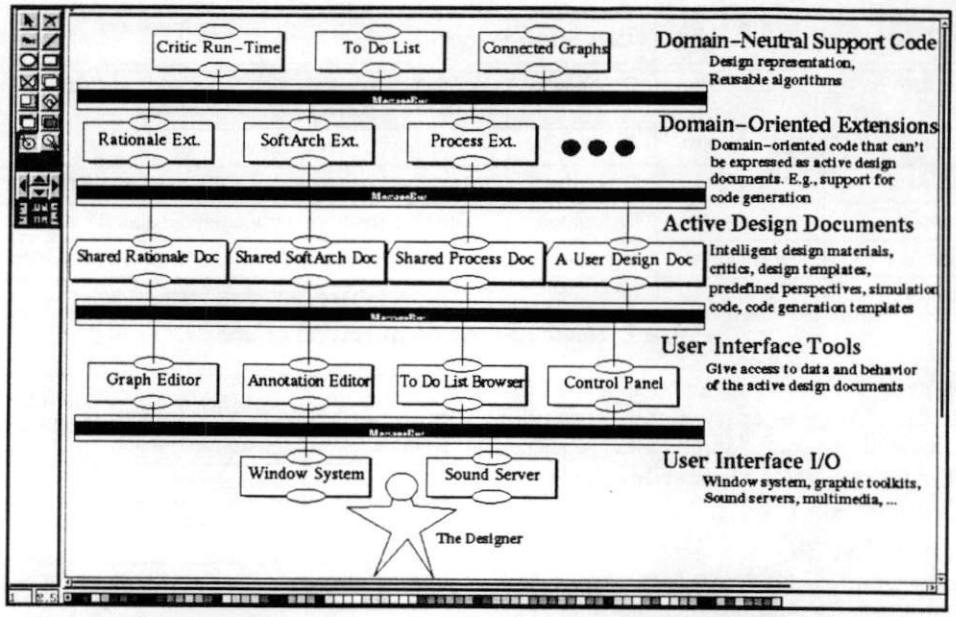

Figure 5. A Screen Shot of Argo Modeling its Own Architecture in the C2 Style (text at right and figure at bottom are unstructured graphical annotations)

software components implements the kernel of the design environment. The second row allows multiple, independent, domain-oriented extensions to the kernel. Each extension defines new facilities if needed, e.g., code generation support in the software architecture extension. The third row contains *active design documents* for the architecture being worked on. These documents are active in that they contain *design materials* (e.g., architectural elements) that carry their own domain knowledge and behavior in the form of critics, simulation routines, and code generation templates. Each extension also provides shared domain-oriented active documents containing a palette of active design materials, reusable design templates, and supporting artifacts. Shared active design documents can be used to share models of software components over the Internet. Individual architectures stored in user design documents may reference (rather than include) code and data in the shared active documents. The fourth row contains user interfaces for architects to access the data and behavior of the active documents. The fifth row of components provides I/O needed to interact with the architect. In C2 style architectures, components in a given row may only send messages requesting operations upward, and messages announcing state changes downward. Between each row of components is a connector that broadcasts messages sent from one side to all components on the other side.

One advantage of this design environment architecture is that artifacts from various supporting domains may be used. For example, domain-oriented extensions for software architecture, design rationale, and process modeling could be active simultaneously. Each of these supporting artifacts may be manipulated, visualized, and critiqued. This provides software architects with first-class supporting artifacts for rationale and process.

In designing this architecture we have attempted to shift away from a large, knowledge-rich design environment that manipulates passive design materials to a smaller, knowledge-poor design environment infrastructure that allows the architect to interact with active design materials. The same trend toward distributing knowledge and behavior to the objects of interest can be observed in the general rise of object-oriented and component-based approaches to software design. Active design materials can be thought of as first-class objects with local attributes and methods. The analysis predicates of critics can be thought of as methods.

The advantages of this shift include increased extensibility, scalability, and separation of concerns in the design environment, and stronger encapsulation of design materials. Extensibility is increased because each architectural element may be packaged with its own attributes and analyses, and thus define its own semantics, which need not be anticipated by the design environment builder. Scalability in the number of critics is increased because there is no need for a central repository of critics — critics simply travel with the design materials. Concerns are separated because the design environment need only provide infrastructure to support analyses packaged as critics, and need not contain any analysis itself. Encapsulation is enhanced because attributes needed for analysis can be made local to the design materials, thus allowing local name spaces and data typing conventions. All of these are important in supporting the evolution of architectures, design environments, and software architecture communities over time.

In Argo, an architecture is represented as an annotated, connected graph made up of nodes, ports, and arcs. Several other architecture description languages (ADLs) are based on similar underlying concepts, including the C2 ADL [19]

and ACME [14]. Figure 6 shows how critics might be associated with architectural elements in a variation of ACME. The C2_Component template includes a sample critic to check a C2 style rule. The Graph_Editor component defines an additional critic to remind architects of a previous experience with that component. Argo defines design material types via Smalltalk and Java classes with private data and access methods. An ADL with that level of encapsulation would be desirable.

Integration with External Analysis Tools

Explicit invocation of external tools scales well in terms of machine resources, but not in terms of human cognitive ability. We believe the cognitive burden of interacting with external tools may be enough to prevent the effective use of such analyses. However, we can address this issue by combining the authoritative and informative approaches.

Existing analysis tools can be repackaged as critics by modifying them to make pessimistic assumptions in cases where exact information is not available in the partial architecture. Alternatively, external batch analysis tools can be paired with "proxy critics" that remind the architect when those tools would be useful. For example, a critic could watch for modifications that affect the result of the batch analysis and check that the architecture is in a state that can be analyzed (i.e., it has no syntax errors that would prevent that particular analysis), and then re-run the batch tool. Here, the critic supports the design process with knowledge about available tools and their applicability to the current partial architecture. Ideally, the output of external analyses should be parsed into individual "to do" list items and linked back to the architecture.

DISCUSSION AND EVALUATION

In this section we discuss and evaluate how well our approach addresses the problems raised in the beginning of the paper. The first four subsections evaluate how well the features of Argo address the identified problems, the fourth focusing on trade-offs between authoritative and informative approaches. The mapping between Argo features and identified problems is summarized in Figure 7. The last subsection deals with empirical evaluation.

The Evolutionary Nature of Architecture Design

The evolutionary nature of the architecture design process suggests the need for early and continuous decision-making support. In Argo, decision-making is supported by critics and the "to do" list.

The effectiveness of a critic in identifying design decisions that need to be made or revised is determined by the specificity of its predicate. The effectiveness of design feedback is determined by the quality and relevance of the supplied information. The critic author is responsible for both the predicate and the information. Our approach cannot guarantee the quality of the author's work, but it does help the architect take advantage of critics from various authors.

Effective decision support demands timely feedback. Critics can positively detect issues very quickly after they are evident in the partial design, typically within seconds of the design manipulation that introduces the problem. Critics can detect problems sooner than authoritative analysis techniques because critics analyze partial architectures and are invoked automatically. However, conflicts between design decisions are often not evident until the last decision is made. Critics can pessimistically predict design conflicts and problems *before* they are evident in the partial design. Criticism control mechanisms help trade early detection for relevance.

```
Template C2_Component () : Component = Component {
    Ports : { top = C2_Port; bot = C2_Port; }
    Properties : {
      ...
      One_Up_One_Down_Critic : Critic {
        Headline : "One Up One Down";
        Type : C2_Style_Rule;
        Decision_Category : checking;
        Analysis_Predicate :
          [ (top connections size) > 1 OR (bot connections size) > 1 ];
        Description : "Instead of connecting multiple message pathways to
a single port, try connecting to a single C2 connector to broadcast messages";
        Expert : "taylor@ics.uci.edu"
      } }

Template T_Graph_Editor( Threads_To_Spawn, Redraw_Method ) :
  Component = C2_Component {
    ...
    Properties : {
      Redraw_Responsiveness_Critic : Critic {
        ...
        Analysis_Predicate :
          [ Threads_To_Spawn < 2 AND Redraw_Method = Display_PS ];
        Description : "The vendor claims one thread is enough, but we
found that using more than one reduced flicker.";
        ...
      } } }
...
System Argo = configuration {
  Components : {
    ...
    Graph_Editor = T_Graph_Editor(3, Bitblt);
  }
  ...
}
```

Figure 6. Specifying Active Architectural Elements

The Cognitive Needs of Architects

Direct manipulation editing and critics support the cognitive needs identified by reflection-in-action. Effective reflection-in-action depends on the architect's ability to incrementally enter a design, see problems in the design, and revise the design. Direct manipulation allows the architect to enter, visualize, and modify the architecture. The use of critics, rather than syntax directed editors, allows the architecture to be entered incrementally while still detecting errors. Critics support reflection by identifying potential problems that the architect is unable to identify because of lack of knowledge or computational effort. The "to do" list and design perspectives work together to focus revision by indicating the relation between feedback and the "offending" parts of the architecture.

The cognitive needs identified under opportunistic design are addressed by critics, the "to do" list, and the process model. Effective opportunistic design depends on the architect's ability to choose which task to do next and to defer tasks that would be better done later. Critics help identify design tasks that need to be (re)done, while the "to do" list gives the architect the visibility and flexibility to (re)order tasks in reaction to the state of the partial design. Furthermore, the "to do" list supports deferred commitment to design details

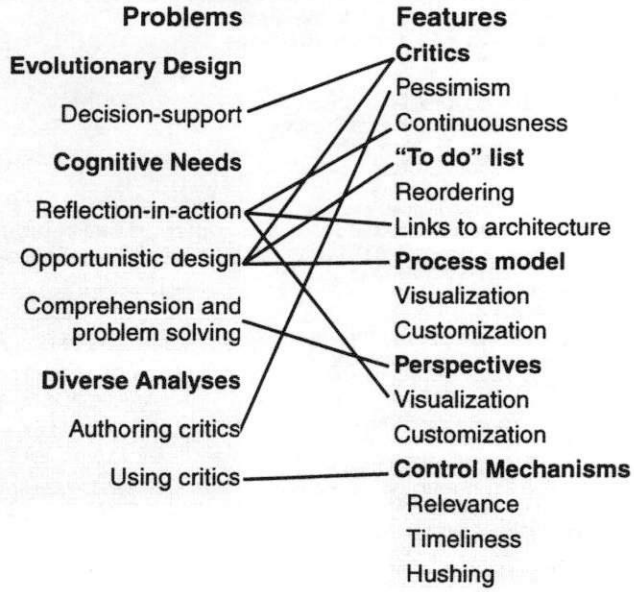

Figure 7. Problem-to-Feature Mapping

by reminding the architect to eventually return to deferred tasks. Argo's process model helps visualize the overall process and (re)orient the architect in that process after a design excursion. Argo primarily informs the architect of which tasks need to be performed. Additional support could be provided by actually estimating the degree of relation between recent decisions and pending design tasks.

Multiple, coordinated design perspectives on the architecture support the cognitive needs identified under comprehension and problem solving. To be effective, architecture design environments must support the construction of perspectives appropriate to project stakeholders and help architects understand the relationships among the contents of different perspectives. Argo supports construction of diverse perspectives with customizable presentation graphics and incremental formalization, as described in [29]. Argo indicates the relationships between perspectives via visual cues such as highlighting all presentations of an architectural element when it is selected in any view.

Support for Diverse Analyses

Pessimism and the informative assumption together reduce the critic author's burden of precision, thus opening opportunities for critic authorship. Critic authors may write predicates that identify potential problems, which is easier than proving the existence of problems. Experts may author speculative critics to test whether a specific piece of guidance actually impacts architects' decisions. In both cases, the architect has the final responsibility for resolving or ignoring the feedback produced.

The "to do" list and informative assumption together allow the architect to make use of diverse (or even conflicting) expert opinions and rules of thumb. For example, one critic could advise that there are too many components at a given level of the architectural decomposition and suggest further hierarchical decomposition, while another might advise that there are too many levels and suggest consolidating existing levels. The architect would use the "to do" list to browse all available feedback relevant to these issues and then make a decision.

Effective use of diverse critics by the architect also depends on the ability to obtain and manage large numbers of critics. In the Java version of Argo, critics are implemented as individual classes and may be dynamically loaded over the Internet. That allows architects to obtain critics authored by diverse, independent experts. To manage large numbers of critics, we associate critics with the definitions of active design materials. Critics are loaded only when those particular design materials are used. This limits the number of critics that are present at a given time. Furthermore, associating critics with design materials allows the producers of software components to supply models of those components with embedded critics. For example, in a software component marketplace, an architect might download several graph editing component models, try them in the current architecture, consider the resulting design feedback, and make an informed component selection.

Authoritative Versus Informative Approaches

This paper has primarily described critics that reside in a design environment and independently perform fairly "inexpensive" analyses. The advantage of these critics comes primarily from the cognitive support they provide. Architects also need "expensive" analytic support, e.g., to detect potential deadlock situations. This subsection evaluates the critic-based approach as a trade-off between in-depth, authoritative analysis and informative, cognitive support, and discusses the combination of the two. The discussion is summarized in Figure 8.

Critics are able to support the cognitive needs of the architect because they are integrated with the architect's decision-making process. This integration demands that critics be dynamically integrated into the design environment so that they have access to the partial architecture as it is being manipulated and to the decision and process models. Critics often provide only "inexpensive" analyses due to the fact that they reside in the design environment and must perform their analyses quickly or incrementally. However, inexpensive analyses need not be trivial or obvious. The value of analysis comes from its impact on the quality of the architecture and the productivity of the architect.

Special purpose analysis tools are not limited by the interaction requirements placed on critics. They can be used for more computationally intensive problems, and often require sophisticated storage and sharing of intermediate results. The time required to perform computationally intensive analyses introduces an unavoidable lag between decision-making and reflection. A much larger barrier to cognitive support is the cognitive burden that external tools place on architects. This cognitive burden is not merely due to user interface oversights, it stems from the lack of design process knowledge in the external analysis tools.

For design decisions that do not interact with others, it is feasible to combine the critic-based approach with

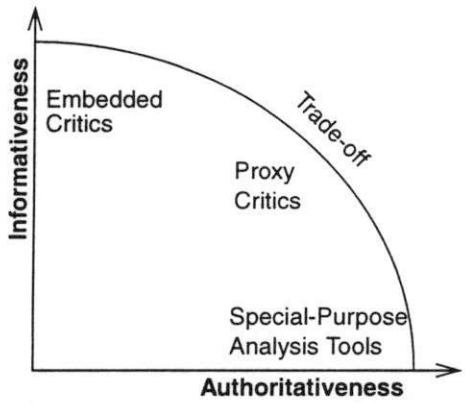

Figure 8. A Space of Analysis Approaches

authoritative analysis. This would provide answers to difficult questions at the time when those answers are needed most. Unfortunately, for most design issues, there are inherit trade-offs that prevent achieving both informative and authoritative feedback. There will always be a gap between the making of a decision and the analysis of that decision. That gap allows the passing of time, expenditure of effort, and loss of cognitive context. When one decision is analyzed in isolation, the gap may be small, but the feedback is at best informative because that decision interacts with others that have not been made yet. When analysis is deferred until groups of interrelated decision have all been made, the gap is necessarily larger, but the feedback may be more authoritative because more interactions are known. When analysis is deferred until after all decisions have been made, feedback may be completely authoritative.

The use of proxy critics (as described in "Integration with External Analysis Tools") enables a choice of points along the trade-off curve. With proxy critics, architects may use authoritative tools without incurring the cognitive burden normally associated with external analysis tools. The external tool performs the main analysis of the architecture, while the critic performs analysis to determine when the main analysis should be applied. This is similar to the separation of concerns that exists between critics and criticism control mechanisms.

Empirical Evaluation

The preceding subsections have provided theoretical evaluation of the critic-based approach. This subsection outlines our current and planned empirical evaluation of Argo as a working tool.

To evaluate Argo's support for scalability and extensibility we will attempt to apply the Argo design environment infrastructure to a variety of application domains. Argo has been used to model C2 style architectures for graphical user interface intensive systems. A project is already underway to apply the Argo infrastructure to object-oriented, real-time systems design. To evaluate how well critics can capture design expertise, we will attempt to cover the guidance provided by other authors in these domains. Capturing C2 design expertise is fairly straight forward since C2 is described as a set of rules that can be implemented as critics. Heuristics on object-oriented and real-time systems design are available from a variety of sources, e.g., [3, 26], which provide a mixture of explicit rules, assumptions, processes, and notations.

To evaluate Argo's support for the cognitive needs of architects, user testing will focus on comparing the productivity of architects using Argo with various features enabled or disabled. Our experiments will focus on:

- the lifespan of design errors when critics are enabled versus disabled;
- the fraction of design errors that are eventually fixed when critics are enabled versus disabled;
- the perceived relevance of design feedback when individual control mechanisms are enabled versus disabled;
- the number of forgotten design ideas when a predefined design process is strictly enforced versus provided as a resource;
- the number of design alternatives explored when various Argo features are enabled versus disabled; and,
- comprehension and recall of a sample design when the architect is allowed to examine it through multiple perspectives versus a single perspective.

The results of these tests should give weights to the arcs in Figure 7 so that we can measure the degree to which identified problems are covered by Argo features. In reaction to this information we may add or revise features.

RELATED WORK

Our focus on the cognitive needs of designers stems from the work of Fischer and colleagues [9, 10] and is motivated by Engelbart's research on augmenting people's ability to solve design problems [5, 7]. In attempting to apply design environments to the domain of software architecture, we have found that the complexity of designing large software systems introduces new challenges that call for new or revised design environment facilities [28]. We extend previous design environment facilities to add support for cognitive needs identified in the cognitive theories of reflection-in-action, opportunistic design, and comprehension and problem solving.

The Hydra design environment [12] (for kitchen floorplans) allows designers to specify design goals as a series of domain-oriented questions and answers; criticism control mechanisms in Hydra activate only critics that are relevant to the specified goals. Argo provides a similar ability to specify architectural goals. The Framer design environment [25] (for GUI window layout) models the design process as a linear sequence of steps; criticism control mechanisms in Framer activate only critics that are relevant to the current step in that process model and prevents the user from moving on to the next process step until outstanding criticism has been resolved. Argo goes

beyond the simple process model of Framer to provide a flexible process model that is more appropriate for software architecture design.

Aesop [13, 33] 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 decision-making process. For example, architectural manipulations that violate style rules may fail without providing any guidance to the architect [33]. In Aesop, most analysis is performed by external tools that are explicitly invoked by the designer. Other software architecture design environments such as DaTE [4] and MetaH [39] also focus on systems-oriented requirements rather than the architect's cognitive needs. Support for systems-oriented requirements is obviously needed; we have assumed that such support will be provided and shifted our focus to cognitive needs.

Research to date has produced a diverse set of authoritative architectural analysis techniques. They include static techniques — such as determining deadlock based on communication protocols between components [2] and checking consistency between architectural refinements [18, 20] — as well as dynamic techniques such as architecture simulation [18]. We hope to combine this research with the critic-based approach to produce tools that provide more complete support for evolutionary architecture design.

SUMMARY AND FUTURE WORK

In this paper we have presented the critic-based approach to software architecture analysis. Argo, our software architecture design environment, follows this approach to support architectural evolution and the architect's decision-making process. Critics continuously supply pessimistic feedback to advise the architect of problematic or pending design decisions. Criticism control mechanisms limit the execution of critics to keep feedback relevant and timely.

We have also presented an architecture that separates reusable design environment infrastructure from active design materials that carry their own analyses. Doing so makes the design environment more extensible and scalable. Associating critics with the architectural elements suggests a way for ADLs to increase encapsulation. Diverse analyses are important for supporting the diverse issues arising in architecture design, and the diverse interests of stakeholders and the software architecture research community. The critic-based approach supports diversity by removing any assumption of cooperation among analysis developers, and by decoupling the architect's cognitive burden from the number of available analyses.

In future work we will continue the themes of our current research. Further identification of the cognitive needs of architects will lead to new design environment facilities to support those needs. We intend to explore the trade-off between the depth and timeliness of feedback. In doing so we will develop a methodology for integrating external analysis tools as discussed above and investigate critics which, over the entire course of the design process, accumulate shared annotations on design materials and perform more in-depth analyses.

It is our goal to develop and distribute a reusable design environment infrastructure that others may apply to new application domains. Successful usage of our infrastructure by others will serve to inform and evaluate our approach. An initial Java version of Argo is available via http://www.ics.uci.edu/pub/arch/.

References

1. Abowd, G., Allen, R., and Garlan, D. Using style to understand descriptions of software architecture. *SIGSOFT Software Engineering Notes*, Dec. 1993, vol.18, (no.5), 9-20.
2. Allen, R. and Garlan, D. Beyond Definition/Use: Architectural Interconnection. *Workshop on Interface Definition Languages*, published in *ACM SIGPLAN Notices*, August, 1994. Vol. 29, No. 8. 35-45.
3. Maher A., Juha K., Jurgen Z. *Object-oriented technology for real-time systems: a practical approach using OMT and fusion*. Prentice Hall, 1996
4. Batory, D. and O'Malley, S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Oct. 1992, vol.1, (no.4), 355-98.
5. Bolcer, G. A. and Taylor, R. N. Endeavors: A Process System Integration Infrastructure, *4th International Conference on Software Process (ICSP4)*, to appear.
6. Engelbart, D. A Conceptual Framework for the Augmentation of Man's Intellect. In: Greif I, ed. *Computer-Supported Cooperative Work: A Book of Readings*. Morgan Kaufmann Publishers, Inc., 1988, 35-66.
7. Engelbart, D. Toward Augmenting the Human Intellect and Boosting our Collective IQ. *Communications of the ACM* 1995; vol. 38, no. 8, 30-33.
8. Fischer, G. Cognitive View of Reuse and Redesign. *IEEE Software*, Special Issue on Reusability 1987; vol. 4, no. 4, 60-72.
9. Fischer, G. Domain-Oriented Design Environments. *Proc. of The 7th Knowledge-Based Software Engineering Conference.* 204-213.
10. Fischer, G., Girgensohn, A., Nakakoji, K., and Redmiles, D. Supporting software designers with integrated domain-oriented design environments. *IEEE Trans. on Software Engineering*, June 1992, vol.18, no.6, 511-22.
11. Fischer, G., Lemke, A., Mastaglio, T., and Morch, A. The role of critiquing in cooperative problem solving. *ACM Transactions on Information Systems*, April 1991, vol.9, no.2, 123-51.
12. Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., and Sumner, T. Embedding Computer-Based Critics in the Contexts of Design. INTERCHI'93. April 1993. 157-164.
13. Garlan, D., Allen, R., and Ockerbloom, J. Exploiting style in Architectural Design Environments. *Proceedings of the Second ACM SIGSOFT Symposium on the*

Foundations of Software Engineering, 1994. Software Engineering Notes, Dec. 1994, vol 19, no.5, 175-88.

14. Garlan, D., Monroe, R., Wile, D. ACME: An Architecture Interchange Language. CMU School of Computer Technical Report CMU-CS-95-219, December 1995.
15. Guindon, R., Krasner, H., and Curtis, W. Breakdown and Processes During Early Activities of Software Design by Professionals. In: G.M. Olson ES S. Sheppard, ed. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ. 1987. 65-82.
16. Kintsch, W. and Greeno, J. G. Understanding and Solving Word Arithmetic Problems. *Psychological Review* 1985;92, 109-129.
17. Kruchten, P. B. The 4+1 View Model of Architecture. *IEEE Software*. Nov. 1995. 42-50.
18. Luckham, D. C., Augustin, L. M., Kenney, J. J., Veera, J., Bryan, D., and Mann, W. Specification and Analysis of system architectures using Rapide. *IEEE Trans. on Software Engineering*. April, 1995.
19. Medvidovic, N., Taylor, R. N., and Whitehead, Jr., E. J. Formal Modeling of Software Architectures at Multiple levels of Abstraction. *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, USA, April 1996.
20. Moriconi, M., Qian, X., and Riemenschneider, R. A. Correct Architecture Refinement. *IEEE Trans. On Software Engineering*, April, 1995, 356-372.
21. Nuseibeh, B., Kramer, J., Finkelstein, A. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. on Software Engineering*, Oct. 1994, vol. 20, no. 10. 760-73.
22. Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, May 1972.
23. Pennington, N. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, Vol. 19, No. 1987, 295-341.
24. Redmiles, D. F. Reducing the Variability of Programmers' Performance Through Explained Examples. *INTERCHI '93 Conference Proceedings*, April 1993, 67-73.
25. Rettig, M. Cooperative Software. *Communications of the ACM*. April 1993. Vol. 36, No. 4. 23-28.
26. Riel, A. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
27. Rist, R. Variability in program design: the interaction of knowledge and process. *The International Journal of Man-Machine Studies* 1990, 1-72.
28. Robbins, J. E., Hilbert, D. M., Redmiles, D. F., Extending Design Environments to Software Architecture Design. KBSE'96, in press.
29. Robbins, J. E., Morley, D. J., Redmiles, D. F., Filatov, V., and Kononov, D. Visual Language Features Supporting Human-Human and Human-Computer Communication. *Proc. of IEEE 1996 Symposium on Visual Languages*. Sept. 1996.
30. Robbins, J. E. and Redmiles D. F. Software Architecture from the Perspective of Human Cognitive Needs. *Proc. of the California Software Symposium*. April 1996.
31. Schoen, D. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books, 1983.
32. Schoen, D. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems* 1992;vol. 5, no. 1, 3-14.
33. Shaw, M., Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
34. Shipman, F. and McCall, R. "Supporting Knowledge-Base Evolution with Incremental Formalization," *Human Factors in Computing Systems, CHI'94 Conference Proceedings*, Boston, MA, 1994, pp. 285-291.
35. Soni, D., Nord, R., and Hofmeister C. Software Architecture in Industrial Applications. *International Conference on Software Engineering 17*, 1995, 196-207.
36. Soloway, E. and Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 1984; vol. 10, no.5, 595-609.
37. Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM* 1988; vol. 31, no. 11, 1259-1267.
38. Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A Component and Message-based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.
39. Verstal, S. Mode changes is real-time architecture description language. In *Proc. of the Second International Workshop on Configurable Distributed Systems*, March 1994.
40. Visser, W. More or less following a plan during design: opportunistic deviations in specification. *International Journal of Man-Machine Studies* 1990; 247-278.