Software Architecture Critics in the Argo Design Environment

Jason E. Robbins and David F. Redmiles
Dept. of Information and Computer Science
University of California, Irvine
Irvine, California, 92697-3425 USA
+1-714-824-7308
{jrobbins,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 that demand a broad range of knowledge of the problem and solution domains. In this paper we describe Argo, a software architecture design environment that supports designers by addressing several cognitive challenges of design. Argo's critiquing infrastructure supports decision making by automatically supplying knowledge that is timely and relevant to decisions at hand. Our discussion centers on a five-phase critiquing process that we use to motivate Argo's features, structure a usage scenario, and characterize related work.

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

1. Introduction

Software architecture is one promising approach to the development of large software systems [21][29][36][38]. Software architectures are high-level design representations of software systems that focus on composition of software components and how those components interact. Software architecture deals with software components, operating system resources, and source code modules. Software architecture relationships deal with communication between components, allocation of operating system resources to components, and dependencies between source code modules and conceptual components.

Ideally, off-the-shelf components are reused, but in practice some components must be customized or developed from scratch [22]. Software architects customize these components by specifying interfaces and choosing values for parameters such as memory allocations, operating system process priorities, and buffer lengths. This component-based approach to development relies on notations that abstract the details of implementation and allow the architect to focus on essential design decisions.

Decision making is an essential activity performed by software architects in designing software systems. The resulting design must satisfy the requirements and not violate constraints imposed by the problem domain and implementation technologies. Requirements are typically stated in terms of needed functionality, limitations on resource use, and non-functional requirements such as extensibility, portability, or scalability. In addition to hard constraints and requirements 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. Even after all constraints are taken into account, the architect has a very large and poorly structured space of possible designs to explore.

Software architects base design decisions 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 be fast but difficult to extend to handle HTML syntax, while another might be more flexible but require a run-time licensing fee. The alternative of developing a new spelling component demands time, budget, and specific technical skills. Architects also need to take into account industry and organizational standards and estimate whether their designs will be understandable and maintainable. Architects acquire the diverse knowledge they need from a variety of sources, both technical and organizational.

In sum, software architectures evolve incrementally as the result of many interrelated design decisions potentially made over extended periods of time. We envision software architecture design as a situated design process in which architects explore paths through a space of alternatives [40]. Particular software architectures can be thought of as a product of one of these paths. Decisions at any point can critically affect

alternatives available later, and every decision has the potential of requiring earlier decisions to be reconsidered.

This paper focuses on the decision-making support provided by critics in Argo, a design environment for software architecture. The next section describes the decision-making challenges faced by software architects. Sections 3 and 4 present a conceptualization of the critiquing process and detailed descriptions of Argo's critiquing features. Section 5 presents a usage scenario showing how these features support software architecture design while keeping the architect in control. Section 6 describes Argo's implementation. Section 7 compares the critiquing approach to other intelligent user interface approaches and compares Argo to other critiquing systems. Section 8 concludes the paper.

2. Problem

In complex domains, no one architect 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 and their own goals and priorities. Even experienced architects need knowledge support in complex domains or when working with unfamiliar design elements. The "thin spread of application domain knowledge" has been identified as a general problem in software development [5]. In fact, it has been worsened by software's newest crisis: a shortage of trained workers [14].

Sound decision making is especially important in the early phases of the software life-cycle. Errors and oversights introduced in high-level design become much more expensive to remove as development proceeds. Typical estimates put the cost of fixing an error in unit testing at more than ten times the cost of fixing the same error in high-level design.

The knowledge that architects use to make design decisions is diverse, heuristic, and tacit. Diverse knowledge is needed to address the variety of design issues in a complex system. Design knowledge is often heuristic because qualities are difficult to measure or the relationship between design choices and qualities is unclear. Design knowledge is tacit; architects cannot articulate or catalog all the knowledge they have, but rather apply it in context [27]. The tacit nature of design knowledge makes it impractical to attempt to build tools that contain complete knowledge.

In addition to the need for knowledge, architects also face cognitive challenges in applying the knowledge that is available to them. The cognitive theory of *reflection-in-action* [33][34] states that designers can best evaluate their designs while they are engaged in making design decisions, not after. Furthermore, availability bias arises when multiple pieces of information influence a decision, but only some of them are readily available. This often results in decisions based on incorrect simplifying assumptions that must be reworked later [39].

Two other cognitive challenges of design are described by the theory of *opportunistic design* and the theory of *comprehension and problem solving*. *Opportunistic design* observes that designers do not follow prescribed design processes. Instead, they choose what to do next as they work through the design, based on cognitive costs [17][42]. Reordering design steps allows designers to follow a train of thought to a satisfactory conclusion, but it can also result in steps being accidentally omitted and substantial overhead spent on process re-orientation.

Comprehension and problem solving addresses the way designers use multiple mental models of the system, each one of which addresses a subset of design issues [20][28]. Using multiple mental models can make each model more understandable, but adds the burden of understanding and maintaining mappings between the models.

We describe the features of Argo that address the cognitive challenges of design raised by these theories in previous work [30]. In the following sections we focus on designers' need for knowledge and how Argo's critiquing infrastructure delivers that knowledge to improve decision making.

3. Approach

The fundamental premise of the critiquing approach is that analysis is most helpful if its results are provided to designers in the context of their decision making. The elements of the critiquing approach are an artifact being constructed, a designer making decisions or choices about that artifact, and timely feedback to the designer about his or her decisions. Thus critiquing systems adhere to the theory of *reflection-in-action*.

This fundamental premise of critiquing systems is shared in part by traditional approaches to software analysis. Critiquing differs from traditional analysis approaches in that the architect's cognitive needs are made central.

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.

The critiquing approach follows 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 and by advising them of possible corrective actions. 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 partially specified architectures. This approach avoids the need to assume that critics have complete knowledge, and facilitates incremental development and improvement of critics.

The approach makes use of *critics*, *criticism control mechanisms*, *feedback management*, *corrective automations*, and *design history*. *Critics* are active agents that support decision making by continuously and pessimistically analyzing a partially specified design. Each critic checks for the presence of a certain condition in the design. Critics are embedded in a design environment where they have access to the architecture as it is being modified. 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 that is not focused on current design decisions. *Criticism control mechanisms* are used to control the execution of critics, so as to inform the architect without distracting from the design task at hand. Since design knowledge is diverse and heuristic, there is likely to be a significant amount of feedback that *is* relevant to current decisions. *Feedback management* tools allow the architect to control the presentation of the design feedback. Presenting design feedback is only useful if it ultimately results in better designs. *Corrective automations* help the architect improve the design by resolving specific, identified problems. Each change to the design may raise new problems or constrain future choices. *Design history* is a timeline of significant design activities, such as problems being introduced or resolved, that may be needed in future decision making.

Each of Argo's critiquing features supports an individual phase of the conceptual critiquing process

Activate → **Detect** → **Advise** → **Improve** → **Record**

Fig. 1. The ADAIR Critiquing Process.

shown in Fig. 1. First, an appropriate subset of all available critics is selected for *activation*. Second, active critics *detect* assistance opportunities. For example, these may be errors that need to be corrected, missing areas of the design that need to be completed, or sub-optimal design choices that should be changed. Third, design feedback items are presented to *advise* the architect of the problem and possible improvements. Fourth, if the architect agrees that a change is prudent, he or she makes changes to *improve* the design and resolve identified problems. Finally, any design changes are *recorded* so that they may be used to inform future decision making. The majority of this paper shows how Argo's features and implementation fit into the ADAIR process. However, Section 7 shows that it also works well for features found in other critiquing systems.

Fig. 2 shows an overview of Argo. The architect uses multiple, coordinated design perspectives to view and manipulate Argo's internal representation of the architecture which is stored as an annotated, connected graph. *Critics* monitor the partially specified architecture as it is manipulated, placing their feedback in the architect's *"to do" list*. Argo's *process model* serves the architect as a resource in carrying out an architecture design process, while the *decision model* lists issues that the architect is currently considering. The decision model is part of Argo's *user model*. The user model also contains information about the architect's preferences and skills. Argo's *goal model* contains information about the desired features of the architecture. *Criticism control mechanisms* use the goal model and user model to ensure the relevance and timeliness of feedback from critics. Argo's "to do" list user interface uses the user and goal

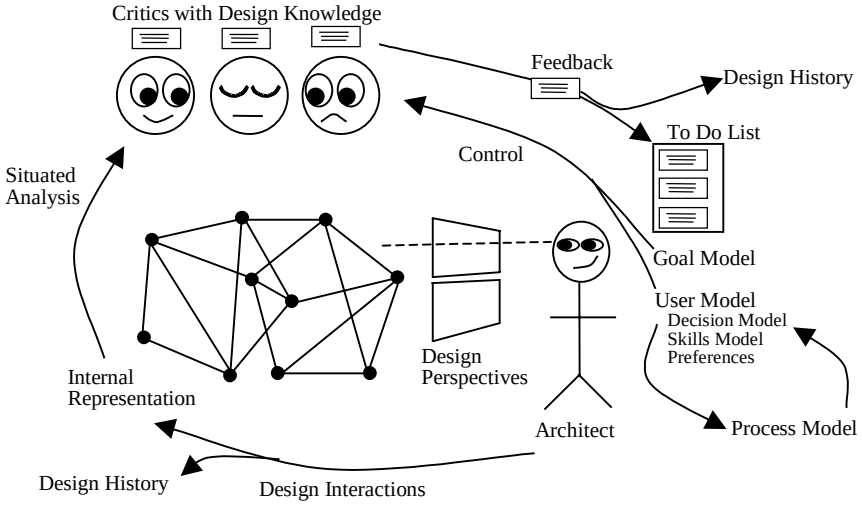

Fig. 2. Overview of Argo.

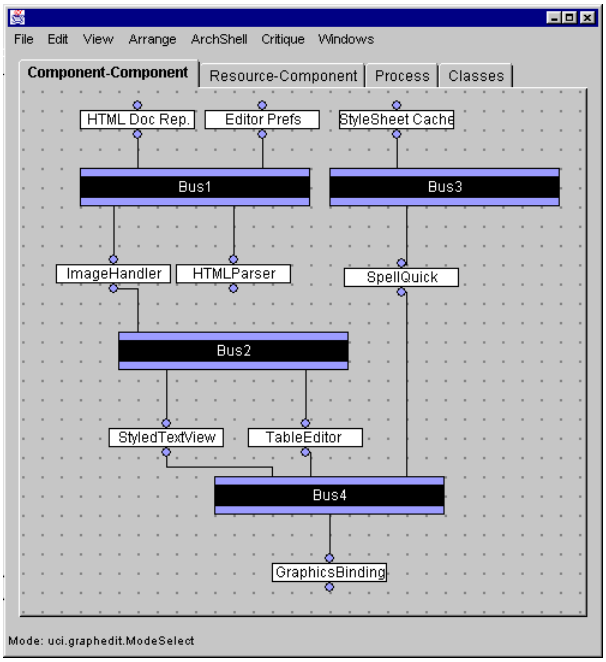

Fig. 3. Screenshot of Argo modeling an example architecture.

models to manage the presentation of outstanding design feedback. Argo's *design history* records all design manipulations and the creation and resolution of each feedback item. Fig. 3 shows a screenshot of Argo modeling an example architecture.

4. Critiquing Features in Argo

4.1. Criticism Control Mechanisms

In the first phase of the ADAIR process, a subset of design critics is selected to be active at any given time. This is done to provide the architect with a usable amount of information. 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 timely and relevant to design decisions being considered by the architect. For example, critics related to maintainability of the architecture should not be active when the architect is trying to concentrate on machine resource utilization. Attributes on each critic identify the types of design goals and decisions that it supports. Criticism control mechanisms check those attributes against Argo's goal and decision models to determine if the critic would produce feedback that is 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 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.

Argo's goal and decision models may be directly manipulated by the architect. Fig. 4 shows Argo's user interface for explicitly declaring the types of decisions that are currently of concern. Also, Argo's decision model is indirectly updated whenever the architect interacts with Argo's process model [30].

4.2. 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 possible improvements 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 can be especially 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. We group critics into types based on the type of domain knowledge that they provide. *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 prompt the architect to consider alternatives to a given design decision. *Evolvability* critics address issues, such as modularization, that affect the effort needed to change

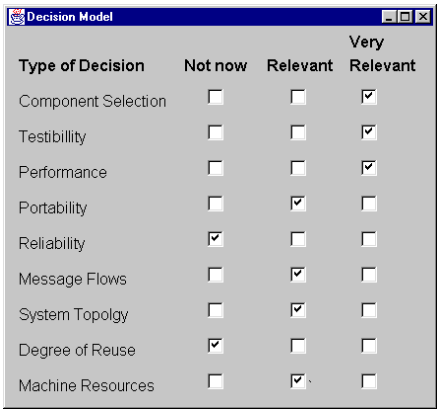

Fig. 4: User Interface for Direct Manipulation of Argo's Decision Model.

Table 1. Selected Example Critics.

Name	*Type*	*Decision*	*Problem Description*
Invalid Connection	Correctness	Message Flows	This component needs the following messages to be sent or received, but they are not present: *<<list of messages>>*
Direct Connection	Correctness	System Topology	Violation of C2 style guidelines. Consider using a message bus to allow new components to be added even after the system is deployed.
Missing Memory Requirements	Completeness	Machine Resources	The memory required to run this component has not been specified. To fix this open the property sheet and enter the amount of memory needed.
Too Many Components	Evolvability	System Topology	There are too many components at the same level of decomposition to be easily understood. Defining a new sub-system can fix this.
Generator Limitation	Tool	Component Selection	The default code generator cannot make full use of this component. Consider using a different component or code generator.
Portability Questionable	Experiential	Portability	Your colleague, *<<name of person>>*, had difficulty using this component under *<<name of OS>>*. You should contact him or her.

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.

Critics produce feedback items consisting of a headline, a brief description, a hyperlink to more detailed information, references to the "offending" design materials, and contact information for the author or maintainer of the critic. The item's headline allows the architect to quickly browse the many feedback items on his or her "to do" list. The description is generated from textual templates explaining the potential problem, why the problem is relevant to design goals, and how the architect might go about fixing it. Each feedback item has references to particular elements that contribute to the problem and are likely to need revision.

Some critics are based on technical considerations and need only be maintained as those considerations change. For example, a syntax critic need only be revised if the architecture description language is changed. However, a broader class of critics deals with the opinions of experts or the experiences of other architects in the organization. For example, an organizational guideline might require all data storage and business logic components to be allocated to hosts that have no user interface components. Contact information, e.g., email addresses, in design feedback provides some of the organizational context needed to resolve problems stemming from mismatches between the design and the design organization.

One difficulty with using critics is that designers are often made uncomfortable by the critic user interface metaphor. The metaphor is that of a critical person always watching over one's shoulder and finding fault with every decision. Designers using systems that follow this metaphor would face constant challenges to their authority, feel the need to guard against criticism, and rarely accept suggestions. Sumner, Bonnardel, and Kallak found that designers using the Voice Dialog Design Environment (VDDE) often changed their behavior to avoid situations where critics might fire and rarely followed the advice of critics [41]. TraumaTIQ lessens this effect by limiting provided feedback and focusing on urgent problems [15]. The Framer design environment attempts to counter the negativity of critics with occasional praise [24]. In building Argo we have tried to mitigate this effect by making critics constructive and by replacing the critic metaphor with that of a dynamic "to do" list. Critics in Argo are constructive whenever possible: their feedback explains the problem, its relevance to stated goals, possible resolutions, and offers corrective automations in some cases. In fact, designers may find that design feedback provides more direct access to these automations than do standard command menus. The dynamic "to do" list metaphor is discussed in the next subsection.

4.3. Feedback Management

Once critics generate design feedback items, those items must be presented to the architect in a usable form without distracting the him or her. In Argo, the "to do" list user interface presents feedback to the architect (Fig. 5). When the architect selects a pending feedback item from the upper pane, the associated (or "offending") design elements are highlighted in all design perspectives and the item's description is displayed in the lower pane. The architect may press buttons to follow a link to relevant background domain knowledge or send email to the person who authored the critic. Together these pieces of information provide a design context that the architect can use in resolving the issue at hand.

The tabs at the top of the "To Do" List window allow the architect to view the "to do" list from alternative perspectives. Each perspective shows a task-specific subset of all feedback items. The first tab shows all outstanding feedback, the second and third show just

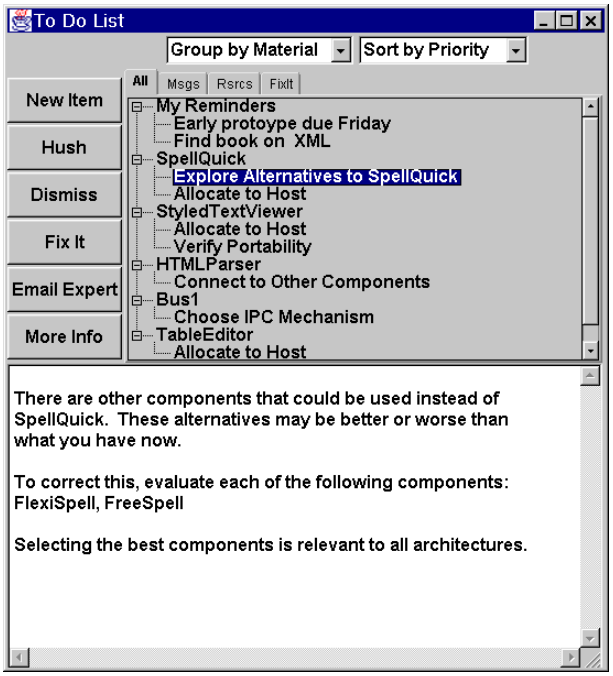

Fig. 5: Argo's "To Do" List User Interface.

feedback relevant to inter-component communication and resource allocation, and the last shows only those items with corrective automations. Within each perspective, items can be grouped and sorted to ease browsing.

Argo's "to do" list almost always has multiple items displayed. Even if there is no outstanding criticism, the architect's personal reminders are still present. Feedback items are continuously being added and removed, and no single addition or removal stands out enough to distract the architect's attention. The architect interacts primarily with feedback items, rather than critics themselves. We expect that architects will find this dynamic "to do" list metaphor both familiar and useful. Designers of complex systems are very familiar with the feeling of having many small tasks pending; the fact that more work could be done is not a challenge to the designer's authority. Argo's "to do" list is useful because it reduces the architect's reliance on short-term memory and provides convenient ways to organize and browse items. A dynamic "to do" list metaphor is also used in the Collaborative Requirements Capture Tool [32].

4.4. Corrective Automation

Often specific problems have specific, automatable solutions. Some critics in Argo provide corrective automation to fix the design problems that they identify. For example, if the architect specifies the memory capacity of a host machine and then assigns software components to that host that exceed its capacity, a critic will identify the problem and offer to fix it in a single step by increasing the specified memory capacity. Alternatively, the architect could reassign some of the software components to other hosts, perhaps under the guidance of a wizard. Corrective automations provide key support for Argo's notion of constructive critics and design history. Argo's use of corrective automations provides a new degree of support that is not found in previous critiquing systems.

Wizards are networks of dialog boxes that lead the user through prescribed steps of a complex operation [6]. Many wizards are linear sequences of requests for information, while others are tree structures that guide the user through a structured set of decisions. In some cases, wizards provide automation that is not available through any other user interface commands. Even if a wizard performs only actions that the user could have performed another way, it still presents a task-specific user interface to these actions and embodies knowledge of how to combine them to achieve the desired result. Providing this knowledge in the form of a wizard reduces the user's need to remember rarely used commands and to plan sequences of low-level manipulations.

4.5. Design History

Because design decisions are interrelated, rationale for past decisions is a key part of the design context of new decisions [23]. For example, an architect building an HTML editing application might initially choose the most full-featured implementation of a table-editing component, only to find that it is incompatible with the spell-checking component. In deciding how to resolve the problem, the architect must know why that particular spell-checking component was used. Blindly replacing the spell-checking component with a more flexible one risks violating the implicit assumptions of related decisions.

Design history is also needed to avoid repeating criticism that is resolved by actions outside of the design environment. For example, if the spell-checking component is a "beta" version rather than a fully tested product, then an organizational critic could advise the architect that all use of beta components require special arrangements with the quality assurance manager. The architect might meet with the manager and agree that it would be all right to use the beta version in this case. The architect would then dismiss the critic's "to do" item, possibly entering a brief rationale. The same criticism should not be presented again for the spell-checking component, despite the fact that the design is in the same state that caused the critic to fire initially.

One challenge faced by design rationale systems is that designers may not make the effort to enter information [16][19][23]. Critics help elicit design rationale as part of the normal design process by acting as foils that give designers a reason to explain their decisions. A recent evaluation of a critiquing system found that experienced designers often explained their

decisions in response to criticism with which they disagree [4].

Aside from building a design history, resolutions to identified problems are sometimes used to adjust Argo's user and goal models. For example, if the system under design is only intended for experimental use, the criticism that beta components require special testing arrangements can be resolved by pressing the "Dismiss" button and choosing "It's not relevant to my goals" (Fig. 6). In response, Argo immediately opens the goal model window so that it may be updated. If the architect chooses "It's not of concern at the moment," Argo opens the decision model window.

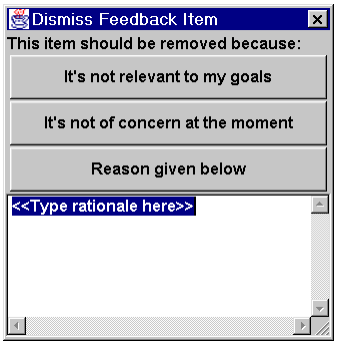

Fig. 6. Argo's Feedback Item Dismissal Dialog.

5. Usage Scenario

In this scenario we demonstrate how Argo's critiquing features support architects in making design decisions. Specifically, the architect is prompted to improve a decision about component selection. To emphasize the fact that the architect maintains control, we structure the scenario as an initial situation and a branching sequence of steps leading to six alternative conclusions. The relationship between these steps is summarized in Fig. 7.

Step 1. The architect is working on an HTML editing tool (Fig. 3). Many components are already chosen, configured, and connected, although some aspects of the functionality of the system have not yet been addressed.

Step 2: Activate. The user and goal models indicate that the architect is willing to consider alternative component choices: i.e., alternative critics are active. The state of the decision model is shown in Fig. 4. Argo activates all critics that are timely and relevant, including an alternative component selection critic.

Step 3: Detect. Argo applies all active critics. An alternative critic detects that the SpellQuick spell-checking component has the same interface as two other library components. This critic determines component substitutability by finding equivalent messages in the components' interfaces. It represents a component's interface as a set of messages that can be sent or received. Messages are considered equivalent if they have the same name and the same number and types of arguments. If all messages used in the current component have equivalents in another, then the critic will suggest considering the other component. Since message semantics are not considered, the architect will need to apply his or her own knowledge. The critic constructs a feedback item and posts it on the architect's "to do" list. The feedback item appears as shown in Fig. 5 and consists of the information shown in Table 2. The fact that the problem was identified is noted in the design history.

Table 2. The Alternative Component Critic's Feedback.

Headline	Explore Alternatives to SpellQuick
Problem	There are other components that could be used instead of SpellQuick. These alternatives may be better or worse than what you have now.
Suggestion	To correct this, evaluate each of the following components: FlexiSpell, FreeSpell
Importance	Selecting the best components is relevant to all architectures.
Offenders	{SpellQuick}
Priority	Normal
Author	jrobbins@ics.uci.edu
MoreInfo	http://www.ics.uci.edu/pub/arch/argo/Alt_Comp_Sel.html
Critic	Alt_Comp_Sel
FixIt	Available

Step 4: Advise. The architect continues working on the architecture uninterrupted. Eventually he or she reaches a reflective point and looks at outstanding design criticism to evaluate the state of the design and decide what should be done next. Browsing the list of "to do" items, the architect sees the headline of the alternative critic's feedback.

Step 5: Advise. The architect reads the problem description and realizes that choosing SpellQuick was fairly arbitrary. Here the architect's understanding of the state of the architecture is improving, even though the design artifact has not changed yet.

Step 6: Improve. In this case, the architect decides to follow the critic's advice and manually performs several manipulations to achieve the desired state. The architect looks at the alternatives and chooses FlexiSpell. He or she deletes SpellQuick, this also delete any relationships between SpellQuick and other components in the architecture. The architect then inserts FlexiSpell, parameterizes it, and connects it to the same components. During this process several other critics may raise or withdraw their criticism as the state of the partially specified architecture changes.

Step 7: Record. Each of the actions performed by the architect is individually added to the design history. Without explicit rationale from the architect, the alternative critic that fired initially can, at best, withdraw its criticism because one of its suggested alternatives

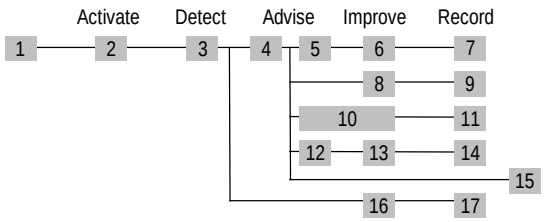

Fig. 7: Forking timeline of scenario steps.

was considered. The same critic is likely to fire again with the criticism that there are still other alternative components that have not been considered.

Step 8: Improve. In this case, the architect looks at the suggested alternative components and decides to keep the current component. This improves the architect's understanding of and confidence in the current design, but it does not modify the design artifact itself.

Step 9: Record. The architect presses the "Dismiss" button to indicate that the matter should be considered resolved. Argo prompts the architect to enter rationale. He or she optionally types a description of why the suggested alternatives were not selected and presses "Reason Given Below" (Fig. 6). A new record is added to the design history with the annotation that it was explicitly resolved by the architect along with the rationale. This particular item will not be posted on the "to do" list again.

Step 10: Advise/Improve. The feedback item references a wizard can be used to swap an alternative component for the current component. In this case, the architect activates the wizard by pressing "Fix It" and works through a sequence of dialog boxes. At each step the wizard explains the step and prompts the architect to make or confirm specific design decisions, such as choosing FlexiSpell as the alternative components to use. The wizard offers to automatically parameterize FlexiSpell with the some of the same parameter values used for SpellQuick and then offers to connect it to some of the same surrounding components. As with manual changes, other critics may raise or withdraw criticism based on the state of the design.

Step 11: Record. Wizards are task-specific user interfaces. Here, the task is the resolution of a specific feedback item. The exact changes made during the wizard interaction can be added to the design history and marked as the resolution to the original criticism.

Step 12: Advise. In this case, the architect reads the design feedback headline and possibly the problem description and decides not to address issues of component selection right now. He or she may simply read past this feedback item and move on to other items. This would leave any other feedback about alternative component choices in the "to do" list.

Step 13: Improve. If the architect wants to focus the "to do" list on other issues he or she may update the decision model to indicate that component choice issues are not of interest at the moment. This action improves the state of the user model. This causes all alternative component choice critics to be deactivated and their criticism withdrawn.

Step 14: Record. The fact that feedback items were withdrawn because of a change in the decision model is recorded in the design history. The new record consists of the new state of the changed part of the decision model and references to the creation records of all feedback items that were withdrawn.

Step 15. In this case, the architect decides that the criticism is currently not of interest, but also declines to take the time to update the user model. Pressing the "Hush" button temporarily disables the critic and hides all "to do" items raised by that critic. After several minutes the critic is automatically re-enabled and its outstanding feedback items reappear on the "to do" list. Hushing is only a feedback management operation, not a step in the critiquing process, since no potential problems are being raised or resolved.

Step 16: Improve. In this case, the architect never looks at the alternative critic's feedback. However, in the normal course of working on the architecture, he or she decides to replace SpellQuick with a new custom component that does not have the same interface as any component in the library. The alternative critic's feedback is automatically withdrawn as soon as Argo determines that it is no longer valid.

Step 17: Record. Argo records each change made to the architecture as the architect works. At some point Argo determines that the alternative critic's feedback is no longer valid. Argo then searches its recent design history for the operations that affected the items offenders. One of those operation records is selected as the resolution to the problem and annotated with a reference to the item's creation record.

In sum, Argo provides a variety of levels of support for improvement and recording design changes. The architect receives feedback about possible improvement opportunities and is free to act on them according to his or her own initiative.

6. Implementation

6.1. Implementation of Criticism Control Mechanisms

Criticism control mechanisms are implemented as Java™ predicates that determine if each critic should be active. Argo provides several criticism control mechanisms, any one of which can deactivate a critic. Preferences in Argo's user model allow groups of critics to be enabled or disabled by type. This allows the architect to control groups of critics easily. Another control mechanism checks the critic's decision types

against those listed in the decision model. This keeps criticism relevant to the tasks at hand. The last predefined criticism control mechanism checks each critic's goal attributes against the goal model. This keeps criticism relevant to the architect's goals.

We have attempted to include a fairly generic set of criticism control mechanisms that can be widely reused. Argo also provides a class framework, source code templates, and examples to aid development of new control mechanisms.

Criticism control mechanisms normally enhance relevance and timeliness. However, these qualities can be reduced if criticism control mechanisms use incorrect information. For example, if the architect mistakenly indicates that criticism supporting machine resource allocation is not of interest, then the architect will see no feedback related to that issue and might assume that the architecture has no resource allocation problems. Argo advises the architect to check the decision model when the "to do" list becomes overly full or if too many "to do" items are being suppressed. The number of suppressed "to do" items is computed by occasionally running deactivated critics without presenting their feedback.

6.2. Implementation of Critics

In Argo, a critic consists of an analysis predicate, goal and decision type attributes, and a feedback item. The stored feedback item contains a headline, a description of the issue at hand, contact information for the critic's author, and a hyperlink to more information. Each feedback item also indicates if its critic provides a corrective automation. We encode analysis predicates using the Java™ programming language. Each critic is associated with one type of target design element and is applied to all instances of that type. Analysis predicates may access the attributes of the target design material and traverse relationships to other design elements. Analysis predicates may also access the user model, goal model, and design history. Critic predicates are written *pessimistically*: unspecified design attributes are assumed to have values that cause the critic to fire. Table 3 presents one critic in detail.

A critic's feedback is produced by copying its stored feedback item and programmatically filling in some of its fields with values specific to the situation. For example, all feedback items have their critic and offenders fields filled in to refer to the critic that created the item and the design materials involved, respectively. Also, many critics customize the headline or short description to include the names of the offenders.

Argo uses two background threads of control to apply criticism and remove invalid criticism without interrupting the user interface thread. Argo's critiquing thread periodically makes a pass over all critics, using criticism control mechanisms to activate or deactivate each of them. Active critics are kept in one of two queues called "hot" and "warm." The hot queue contains only critics that are likely to fire soon. Argo promotes a critic from the warm queue to the hot queue when the architect performs a design manipulation that is likely to cause that critic to fire. Likelihood is encoded as a mapping from design manipulations to critics. For example, if the architect changes the interface of a component, then the correctness critic that checks for interface mismatches would be promoted. The critiquing thread applies each critic in the hot queue and then demotes it to the warm queue. If the hot queue is empty, critics from the warm queue are applied and placed back in the warm queue.

Table 3. The Alternative Component Critic in Detail.

Name	Alt_Comp_Sel
Goals	Any (This critic is always relevant)
Decisions	{ Component_Selection }
Type	Alternative Critic
Target	Software Components
Predicate	Let C = the target software component. Let USED_MSGS = messages sent or received by C. If USED_MSGS is empty then return NO_PROBLEM. Let ALTS = the set of all library components that implement all messages in USED_MSGS. If ALTS is empty then return NO_PROBLEM, else return PROBLEM_DETECTED.
Feedback	(see Table 2)
Corrective Automation	Component Replacement Wizard

Encoding a complete mapping from design manipulation to critics that are likely to fire after that manipulation would result in a more efficient implementation. However, this would increase the effort needed to define and maintain critics and manipulations. Argo's current implementation eventually runs all active critics, even without any mappings. Partial mappings may be defined and yield incremental increases in efficiency.

Argo's invalid feedback removal thread periodically makes a pass over the architect's "to do" list and checks to see if each item is still valid. A feedback item is still valid if the critic that created it would do so again. By default, the critic is reapplied and the resulting feedback item is compared to the one in the "to do" list, however critics may override this default behavior to do more efficient validation.

Argo provides a class framework, source code templates, and examples to aid critic implementers. Authoring a new critic entails selecting a starting template, filling in relevance and timeliness attributes, coding an analysis predicate, and writing a headline and brief description. The fact that critic are pessimistic makes coding analysis predicates somewhat easier since pessimistic assumptions can replace some conditionals. If the critic is deployed and practicing architects email complaints that the critic fires too often, then the

pessimistic assumptions may be refined or replaced with authoritative tests. In this way, critics may be incrementally improved over their life-cycle by adding more precise or up-to-date domain knowledge [10][11].

6.3. Implementation of Feedback Management

Argo's "to do" list user interface involves four distinct windows. The "To Do" List window allows the architect to browse pending items and read their descriptions (Fig. 5). Perspectives on the "to do" list are implemented as predicates that select a subset of all pending items to be displayed. Grouping rules organize the items by offending element, critic, goal, or decision. Each item is shown in all applicable groups. Items may be sorted by time order, priority, or difficulty. Priority is estimated based on the priorities specified in the goal model. Difficulty is estimated based on the skills needed to resolve the problem and the architect's self-evaluation in the skills model. The Dismiss Feedback Item dialog prompts the architect for the reason an item is being dismissed (Fig. 6). The New Item dialog allows the architect to enter a personal reminder.

The Email Expert dialog allows the architect to send an email message to the critic's author or maintainer. By default, the message includes the selected feedback item as a starting point for discussion. In future work we will investigate how organizational memory techniques can be used to enhance discussions between experts and practicing architects [1].

6.4. Implementation of Corrective Automation

Argo represents corrective automations programmatically as Java classes. Critics contain a reference to the corrective automation class, if one is provided. A method of the automation class is invoked when the user selects a critic's feedback item and presses the "Fix It" button.

We currently provide support for authoring corrective automations only in the form of source code examples. In designing critics and wizards, we have found diagrams like the one shown in Fig. 8 useful. Nodes with rounded corners apply predicates to the design; rectangular nodes present dialog boxes to the architect, parallelogram nodes perform design manipulations. Each node is parameterized with details of the test, dialog, or manipulation. The flow of control proceeds from left to right, following all branches, and passing through predicates that evaluate to true. If the flow of control reaches the bull's-eye node then that critic fires and produces a design feedback item with the description shown. If the architect presses the "Fix It" button, control continues out of a bull's-eye, possibly into a sequence of dialog boxes. The critics shown in Fig. 8 are from the object-oriented design domain. In future work we will investigate using these networks as Argo's internal representation of critics and wizards.

6.5. Implementation of Design History

Argo represents design history as a time ordered sequence of records that may contain references to earlier records. Three types of records are supported: feedback creation records, manipulation records, and comment records. All records are marked with the date and time of recording and the login ID of the architect. A feedback creation record contains all the information found in a feedback item. A manipulation record contains a description of the change preformed and the new value of the affected part of the design. For example, changing the memory capacity of a host would create a record including the new memory capacity. Comment records contain an unstructured comment entered by the architect. For example, explicitly dismissing a feedback item and then pressing the "Reason Given Below" button generates a comment record. When a manipulation or explicit dismissal resolves an outstanding feedback item, a reference is added from that record to the item's creation record.

Currently Argo uses the design history only to ensure that previously resolved items are not presented again. An additional index data structure is used to facilitate look-ups by critic type and design material instance. In future work we will investigate design history browsing, search, and visualization.

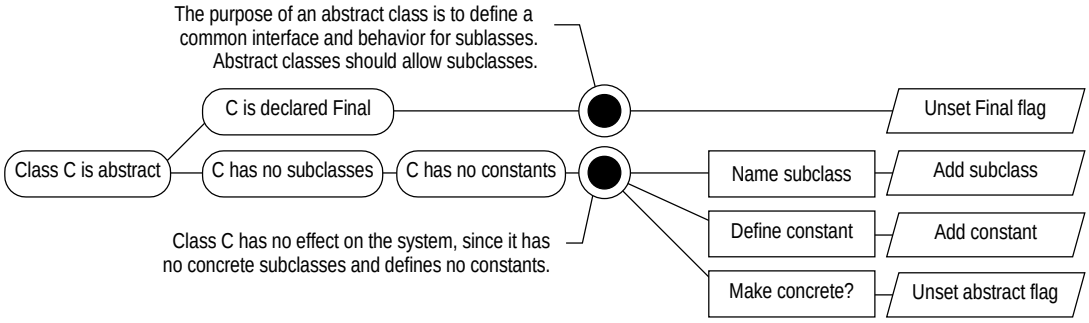

Fig. 8. Diagrammatic Representation of Two Critics with Four Corrective Automations.

Table 4. Related intelligent user interface approaches.

	Activate	*Detect*	*Advise*	*Improve*	*Record*
Tutors	●●●	●●●	●●●	●●	●●
Critics	●●●	●●	●●●	●	●
Wizards		●	●●	●●●	●
Dynamic Revision		●●●		●●●	
Concurrent Checking		●●●	●●	●●●	
Coaches		●●●	●●●	●●●	●
Decision Networks	●●	●●●	●●●	●●●	●
Interface Agents	●●●	●●	●●	●●	●●

7. Related Work

In this section we compare the critiquing approach to other intelligent user interface approaches and then compare Argo to other critiquing systems. *Intelligent user interface* systems structure or modify their interface elements based on implicit or explicit models of the user and/or the user's task. *Critiquing systems* combine critics, feedback management, and support for critic authoring. Critiquing systems are normally part of *design environments* that provide features for design construction, design visualization, and design process support.

7.1. Related Intelligent User Interface Approaches

Table 4 characterizes several intelligent user interface approaches with respect to the phases of the ADAIR critiquing process shown in Fig. 1. We discuss each of these approaches below.

Tutoring systems, such as Geometry Tutor [2], have used detailed user models to implicitly activate tutors and to customize the advice that tutors give to students. Tutoring systems evaluate a student's work and provide feedback to help the student achieve specific lesson objectives. Tutoring systems are constructive in that they give hints to students that are having difficulties with the assigned task. Recording can be performed for the benefit of the instructor, but the most direct use of recording is to update the user model as the student makes progress. Tutors differ from critics in that tutors are intended to help students with artificial educational exercises while critics aid professional designers in actual work situation. Tutors must be programmed with detailed knowledge of the objectives of the exercise and overall lesson plan. Furthermore, they must posses or be able to generate, a complete solution. This requirement limits application of tutoring systems to very well understood domains. In contrast, critics must function with partial knowledge as the problem and solution co-evolve [8].

Critiquing systems, such as in Janus [8], use a partial specification of a problem in choosing which critics should be activated. Critics detect problems in the proposed solution and provide advice to the designer, but most do not play an active role in the improvement of the design or the recording of design activities. We discuss eight critiquing systems below.

Wizards, such as TaskGuide [6] and those found in Microsoft Office [26], are user interface dialogs that guide the user through a sequence of steps or decisions. Wizards are not concurrent processes and need not be activated by the system. Detection of assistance opportunities is the user's responsibility. The primary strength of wizards is their procedural knowledge that allows them to explain and perform the steps needed for specific design changes. For the most part, recording in wizards is limited to the ability to backtrack through steps to revise information; however, some wizards record decisions in the artifact being designed.

Dynamic revision is a feature commonly found in word processors such as MS Word and text editors such as Emacs. These systems watch what users type and automatically replaces some strings with others. For example, "teh" is replaced with "the," and "HEllo" is replaced with "Hello." Concurrent checking is a similar feature that automatically highlights spelling errors as they are typed and offers suggested corrections through a pop-up menu. Both approaches limit user modeling to customizable dictionaries and require explicit activation, but are strong on detection and improvement in the limited domain that they address. Dynamic revision is effective at making improvement, however since it does not advise the user or ask for confirmation, these improvements are sometimes undesired. In fact, it is most noticeable when it has performed an undesired modification. Concurrent spell checking, in contrast, does visually advise the user and ask for confirmation before making a change. This takes extra effort on the part of the user, but gives a sense of control.

Coaching systems, such as COACH [35] and Lumière (Microsoft Office Assistant) [18][26], assist users by watching their actions and suggesting help topics. The primary difference between this approach and the critiquing approach is that coaches assist in tool use while critics assist in design decisions. COACH is a dedicated system that is always activated. Lumière is also always active, although it may be hidden, in which case it only reappears to provide high priority feedback. COACH detects assistance opportunities with rules and an adaptive user model, while Lumière uses a Bayesian network. Feedback primarily consists of a set of help topics or suggested actions to perform next. Feedback presentation is more intrusive in coaching systems than in critiquing systems because the goal of teaching tool use demands immediate feedback when breakdowns are detected. Improvement is well supported with single step corrections or wizards. COACH records interactions by updating its user model.

Decision networks [37] are related groups of agents that together provide support for most phases of the ADAIR process. A dialog generation component activates appropriate agents based on a task model. Influencers provide tutoring before or during a task. Debiasers provide negative feedback when mistakes are made, as do most critics. Clarifiers present feedback to designers in graphical or textual form. Directors provide task-specific support for carrying out improvements, as do wizards. Silverman and Mezher suggest that debiasers should learn from interactions with designers; specifically, it should suppress criticism that the designer had previously rejected [37].

Interface agents assist users primarily by continuously retrieving or filtering information. For example, interface agents can sort and prioritize one's email, filter news streams, or recommend entertainment [25]. Agents are implicitly activated based on models of the user and goals. Once an improvement opportunity is detected, agents may either advise the user of the opportunity or, if confidence in the goal model is high, take immediate action. Agents do not form histories, however they do learn rules and refine their user and goal models by analyzing interactions with users and other agents. Learning agents are most effective when the user performs repetitive actions and are most useful when personalized to a particular user. Knowledge-based approaches are more suitable design support where user may lack needed knowledge.

7.2. Critiquing Systems

In Table 5 we characterize seven critiquing systems according to the phases of the ADAIR process. We do not attempt to evaluate the knowledge embedded in each system, instead we focus on how that knowledge is applied, presented, and ultimately used to improve the design. Most of these are subsystems of integrated design environments, while others are stand–alone tools. Each system primarily uses either *comparative critiquing* or *analytic critiquing*.

Comparative critiquing supports designers by pointing out differences between the proposed design and a design generated by alternative means, for example a planning system with extensive domain knowledge. In this respect, comparative critiquing systems are similar to tutoring systems and suffer from the same limitation. Pointing out differences can lead designers to make their design more like the generated design or cause them to re-examine their reasons for making different decisions.

Analytic critiquing uses rules to detect assistance opportunities, such as problems in the design. This aids designers by guiding them *away* from recognized problems rather than guiding them *to* known solutions. In general, analytic critics can be built incrementally and applied throughout in the design process. Substantial domain knowledge is needed to implement analytic critics, but they need not have access to a generated solution. This allows analytic critics to be applied to a broader range of domains.

Table 5. Comparison table for critiquing systems.

	Activate	*Detect*	*Advise*	*Improve*	*Record*
TraumaTIQ	●●	comparative	●●●		
CLEER		analytic	●●		
UIDA	●	both	●	●●	●
Janus	●●	analytic	●●	●	●
Framer	●●●	analytic	●●	●●	●
VDDE	●	analytic	●●	●	●
Argo	●●●	analytic	●●●	●●●	●●

TraumaTIQ is a stand-alone system that critiques plans for treatment of medical trauma cases, such as gunshot wounds [15]. One emphasis of TraumaTIQ is the time-critical nature of its domain. TraumaTIQ analyzes the doctor's treatment plan to infer its goals and then compares it with one generated automatically. The advice given is focused on concisely achieving *communicative goals* based on the urgency and severity of plan differences. Advice is in the form of English text generated from templates and a domain-specific language model. TraumaTIQ does not provide automated improvement of the treatment plan, nor does it record criticism resolutions.

CLEER is integrated into a computer aided design system for placement of antennas on military ships. Critics in CLEER check constraints dealing with mechanical and electromagnetic features of the design. CLEER presents negative feedback and does not constructively aid the designer in improving the design or recording a design history. Silverman and Mezher propose an enhanced version of CLEER that would use decision networks to add support for activation, advisement, and improvement [37].

The User Interface Design Assistant (UIDA) is a stand-alone system that critiques user interface window layouts for compliance with Motif style guidelines and consistency with other window layouts in the same application [3]. UIDA has no user model, but the designer may explicitly activate groups of style rules. UIDA performs computational critiquing by applying style rules and performs comparative critiquing by recording and comparing the particular set of rules satisfied by each layout. Comparative critiquing of this type does not require a generated solution. Relative to other systems reviewed, UIDA is weak in advisement and strong in improvement. Advice is limited to a sequence of brief prompts asking the designer to confirm suggested changes. Corrective automations are provided with the rules and result in a new version of the window layouts at the end of the critiquing session. A history of rule applications is kept only for the duration of the session to support comparing different layouts.

The Janus family consists of successive versions of a household kitchen design environment, named Crack, Janus, Hydra, and KID [8][12][13]. A goal model implicitly activates critics relevant to stated design goals. Goal specification sheets prompt the designer to provide information through a structured set of choices. Although similar to wizards, specification sheets primarily affect the goal model rather than the design itself. Furthermore, designers using Hydra can select a critiquing perspective (i.e., critiquing mode) to activate critics relevant to a given set of design issues and deactivate others. These design environments use automatically applied computational critics that produce brief problem descriptions and links into a hypermedia argumentation database. In KID, feedback items are sorted by priority. The argumentation database consists of arguments taking various sides of domain-oriented design issues. Janus's critics primarily advise the designer of problems, while the argumentation describes both problems and possible solutions. Corrective automations are not supported. Decisions can be recorded as additions to the argumentation database, but these are not interpreted by the system and do not represent the history of a single design.

The Framer design environment for user interface window layout is much like the ones in the Janus family [24]. However, Framer uses a "to do" list interface metaphor that groups feedback according to suggested steps in a prespecified process. This process model is also used to activate critics when timely. Furthermore, Framer strengthens support for improvement by associating corrective automations with feedback items.

VDDE is a design environment for voice dialog systems, i.e., nested function menus accessed via a telephone [30]. VDDE does not have a user or goal model, instead designers directly specify which sets of critics should be active, their priorities, and how actively they should be applied. Unlike Hydra, multiple sets of critics can be active simultaneously. VDDE's advice is prioritized and consists of a brief description of a problem, a set of offending design materials, and a link into an argumentation database. Problem descriptions are generated from textual templates that are filled in with details of the problem situation. VDDE does not provide corrective automations or phrase its criticism in constructive terms. Support for recording design decisions is similar that found in the Janus family. Sumner, Bonnardel, and Kallak describe a model of VDDE's critiquing process that centers on detecting and advising the designer of problems [4][41].

We have built on the previous work described above and implemented specific support for each phase of the ADAIR critiquing process. Argo has both a user model and a goal model to support activation. Argo's decision model is automatically updated when the architect works with Argo's process model, however Argo does not infer goals from the partially specified design as does TraumaTIQ. Argo's critics analyze the design and produce feedback items with more kinds of design context than those produced by other systems; specifically, it provides contact information for relevant experts and stakeholders. Feedback management in Argo is more flexible than that of other systems reviewed. Improvement is supported by constructive advice and corrective automations, which may take the form of wizards. Argo records design activities done in the environment and some of their relationships, but it does not yet make much use of this information.

8. Conclusions

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. Our general approach is to find cognitive theories that identify these needs and implement supporting features in Argo. In this paper we described Argo's decision-making support as applied to the software architecture design domain. Argo's critiquing features support a critiquing process consisting of five phases: Activate, Detect, Advise, Improve, and Record. We have found the ADAIR critiquing process useful in building, describing, and evaluating Argo.

Acknowledgements

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research 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, Air Force Research Laboratory or the U.S. Government.

References

[1] Ackerman M, Augmenting the organizational memory: a field study of answer garden. CSCW'94. October 1994. Chapel Hill NC, USA. 243-252.

[2] Anderson J R, Corbett A T, Koedinger K R, and Pelletier R, Cognitive tutors: lessons learned. Journal of Learning Sciences. 4 (1995). 167-207.

[3] Bolcer G A, User interface design assistance for large-scale software development. Automated Software Engineering. 2 (1995). No. 3. 203-217.

[4] Bonnardel N and Sumner T, Supporting evaluation in design: the impact of critiquing systems on designers of

different skill levels. Acta Phychologica. 91 (1996). 221-244

[5] Curtis W, Krasner K, and Iscoe N, A field study of the software design process for large systems. Comm. ACM. 31 (1988) no. 11. 1268-1287.

[6] Dryer D C, Wizards, guides, and beyond: rational and empirical methods for selecting optional intelligent user interface agents. IUI'97. Orlando FA. January 1997. 265-268.

[7] Fischer G, Domain-oriented design environments, Proc. Seventh Knowledge-Based Software Engineering Conference. (1992) 204-213.

[8] Fischer G, Girgensohn A, Nakakoji K, and Redmiles D, Supporting software designers with integrated domain-oriented design environments. IEEE Trans. Software Engineering. 18 (1992). No. 6. 511-522.

[9] Fischer G, Lemke A C, McCall R, and Morch A I, Making argumentation serve design. Human-Computer Interaction. 1 (1991) 393-419.

[10] Fischer G, Lindstaedt S, Ostwald J, Stolze M, Sumner T, and Zimmerman B, From domain modeling to collobative domain construction. Symposium on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS'95). Ann Arbor MI. August 1995. 75-85.

[11] Fischer G, McCall R, Ostwald J, Reeves B, and Shipmann F M III, Seeding, evolutionary growth and reseeding: supporting the incremental development of design environments. Proc. Computer-Human Interaction 1994 (CHI'94). Boston MA. 292-298.

[12] Fischer G, Nakakoji K, and Ostwald J, Supporting the evolution of design artifacts with representations of context and intent. Symposium on Designing Interactive Systems: Processes, Practices, Methods, and Techniques (DIS'95). Ann Arbor MI. August 1995. 7-16.

[13] 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.

[14] Fox R, News track, Comm. ACM. 40 (1996). no. 5, 9-10.

[15] Gertner A S and Webber B L, TraumaTIQ: online decision support for trauma management. IEEE Intelligent Systems. January/February 1998. 32-39.

[16] Grudin J, Why CSCW applications fail: problems in the design and evaluation of organizational interfaces, Proc. Conf. on Computer-Supported Cooperative Work (CSCW'88), ACM, New York, Sept. 1988, 85-93.

[17] 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. 65-82.

[18] Horvitz E, Agents with beliefs: reflections on bayesian methods for user modeling. Proc. Sixth International Conference on User Modeling. Chia Laguna, Sarinia. June 1997.

[19] Jarczyk A P J, Loffler P, and Shipmann F M III, Design rational for software engineering: a survey. Proc. Twenty-fifth Hawaii Inter. Conf. on System Sciences, 1992.

[20] Kintsch W and Greeno J G, Understanding and solving word arithmetic problems. Psychological Review. 92 (1985) 109-129.

[21] Kruchten P B, The 4+1 view model of architecture. IEEE Software. Nov. 1995. 42-50.

[22] Kruger C W, Software reuse, Computing Surveys. 24 (1992) no. 2, 131-183.

[23] Lee J, Design rationale systems: understanding the issues. IEEE Expert. (1997). 78-85.

[24] Lemke A C and Fischer G, A cooperative problem solving system for user interface design. Proc. Eighth National Conference on Artificial Intelligence (AAAI'1990). 479-484.

[25] Maes P. Agents that reduce work and information overload. Comm. ACM. 37 (1994). No. 7. 31-40.

[26] Microsoft Corporation, Getting results with Microsoft® Office 97, 1997.

[27] Palanyi M, The tacit dimension, Doubleday, Garden city, NJ, 1966.

[28] Pennington N, Stimulus structures and mental representations in expert comprehension of computer programs. Cognitive Psychology. 19 (1987). 295-341.

[29] Perry D and Wolf A, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes. 17 (1992) no. 4, 40-52.

[30] Repenning A and Sumner T, Using agentsheets to create a voice dialog design environment, Proc. SAC'92, vol. 2, ACM Press, 1199-1207.

[31] Robbins J E, Hilbert D M, and Redmiles D F, Extending design environments to software architecture design. J. Automated Software Engineering. To appear, 1998.

[32] Rogers I, The user of an automatic "to do" list to guide structured interaction. CHI'95 conference companion. Denver CO. 232-233.

[33] Schoen D, Designing as reflective conversation with the materials of a design situation. Knowledge-Based Systems. 5 (1992) no. 1. 3-14.

[34] Schoen D, The Reflective Practitioner: How Professionals Think in Action. 1983. New York: Basic Books.

[35] Selker T, Coach: a teaching agent that learns. Comm. ACM. 37 (1994) no. 7.

[36] Shaw M and Garlan D, Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[37] Siverman B G and Mezher T M, Expert critics in engineering design: lessons learned and research needs. AI Magazine. Spring 1992. 45-62.

[38] Soni D, Nord R, and Hofmeister C, Software architecture in industrial applications. Proc. Inter. Conf. on Software Engineering 17 (1995). 196-207.

[39] Stacy W and MacMillian J, Cognitive bias in software engineering. Comm. ACM. June 1995. 57-63.

[40] Suchman L A, Plans and Situated Actions, Cambridge University Press, Cambridge, UK, 1987.

[41] Sumner T, Bonnardel N, and Kallak B H, The cognitive ergonomics of knowledge-based design support systems. Proc. Computer-Human Interaction (CHI'97).

[42] Visser W, More or less following a plan during design: opportunistic deviations in specification. Int. J. Man-Machine Studies. 1990. 247-278.