Extending Design Environments to Software Architecture Design

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

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

Information & Computer Science
University of California, Irvine
Irvine, CA

Abstract

Domain-oriented design environments are cooperative problem-solving systems that support designers in complex design tasks. In this paper we present the facilities and architecture of Argo, a domain-oriented design environment for software architecture. Argo's architecture is motivated by the desire to achieve reuse and extensibility of the design environment. It separates domain-neutral code from domain-oriented code, which is distributed among intelligent design materials as opposed to being centralized in the design environment. Argo's facilities are motivated by the observed cognitive needs of designers. These facilities extend previous work in design environments to support reflection-in-action, opportunistic design, and comprehension and problem-solving.

Keywords: Domain-oriented design environments, critics, software architectures, architectural styles, human-computer interaction, human cognitive skills.

1. Introduction

Domain-oriented design environments (DODEs) are cooperative problem-solving systems that support designers in complex design tasks [6]. They are domain-oriented in that important concepts and constructs in a particular domain are built directly into the environment. They are cooperative in that they take into account the complementary strengths and weaknesses of humans and computers. Domain orientation helps close the gap between designers' knowledge and the notation used by the environment. Cooperative problem-solving lets designers focus on specifying and adjusting design goals, decomposing problems into subproblems, and so on, while the computer supports designers by providing external memory, hiding non-essential details, checking for inconsistencies or potential design flaws, and providing basic design guidance, analysis, and visualization capabilities [19].

Domain-oriented design environments have been recognized as complementary to more traditional approaches to knowledge-based software engineering [6]. In contrast to program synthesis-oriented approaches, DODEs provide a more interactive, iterative model that takes into account the evolutionary nature of design and the cognitive needs of designers. Motivated by the potential benefit of using design environments to support the needs of software architects, and, more generally, by the appealing arguments of augmenting people's ability to solve design problems [3, 4], we have built a software architecture design environment, called Argo[1], to support the design of complex software systems.

The design environment facilities explored by Fischer and others [6-10] have provided an essential basis for our work. In building Argo, however, we have found it necessary to extend the basic facilities provided by these earlier systems, and have devised an architecture that will support further extensions and applications to new domains. More specifically, our facilities extend previous work to more fully support, and support in a more integrated fashion, the cognitive needs of designers as identified by the cognitive theories of *reflection-in-action*, *opportunistic design*, and *comprehension and problem-solving*. Furthermore, our architecture motivates a shift from a large, knowledge-rich design environment that manipulates passive design materials to a smaller, knowledge-poor design environment infrastructure that allows the user to interact with intelligent design materials.

In Section 2, we briefly survey previous work on design environments and identify key concepts and motivations. In Section 3, we discuss architectural and representational

This research is supported in part by the Air Force Material Command and the Advanced Research Projects Agency under Contract Number F30602-94-C-0218, and by the National Science Foundation under Contract Number CCR-9624846. Additional support is provided by Rockwell International. The content of the information does not necessarily reflect the position or the policy of the funders and no official endorsement should be inferred.

1. Argo was the name of the ship that the Argonauts sailed in Greek Mythology. We hope that our Argo will aid software architects in navigating design spaces.

issues that have allowed us to extend these themes. We present our ideas in detail so other researchers can apply them in building their own design environments. In Section 4, we describe our extended facilities for supporting the cognitive needs of designers, and motivate our claims with reference to cognitive theories of design. In particular, we discuss how design feedback facilities support reflection-in-action, how a design process model and a "to do" list support opportunistic design, and finally, how multiple, coordinated design perspectives support comprehension and problem-solving. In each case, we explain how these facilities are implemented in Argo. We conclude with a summary of our contributions and future work.

2. Previous work on design environments

A domain-oriented design environment [7] is a tool that augments a human designer's ability to design complex artifacts. The concept of human augmentation is based on the work of Engelbart [3, 4] and others who researched ways computers could help enhance peoples' performance of intellectual tasks. Design environments must address systems-oriented issues such as design representation, transformations on those representations (e.g., generating code from a specification [11, 15]), and application of analysis algorithms. Furthermore, they go beyond most tools in their support for the designer's cognitive needs [22].

The cognitive theory of reflection-in-action [23, 24] 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. A similar process can be observed when modifying an existing design.

Design environments support reflection-in-action by using critics to give feedback on the design. Critics are agents that watch for specific conditions in the partial design as it is being constructed and notify the designer when those conditions are detected. Critics can be used to deliver knowledge to designers about the implications of, or alternatives to, a design decision. In the vast majority of cases, critics simply advise the designer of potential errors or areas needing improvement in the design; only the most severe errors are prevented outright, thus allowing the designer to work through invalid intermediate design states. Designers 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 design. Often feedback on issues that the designer had overlooked or was unaware of is the most valuable feedback.

Designers can benefit from domain knowledge when it is delivered to them via critics. Examples of domain knowledge include 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

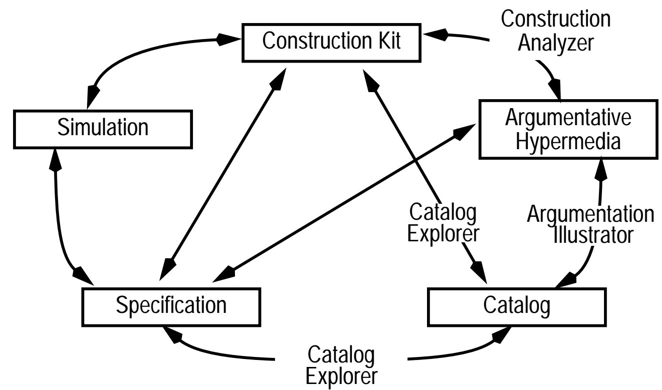

Figure 1: Design environment facilities of Janus (adapted from [6])

rules, and the (potentially conflicting) opinions of domain experts. Rather than place all the burden of precision and restriction on the critic authors, design environments assume that the designer is capable of making final decisions regarding the application of the feedback given.

We can define a variety of potential types of critics, and each type delivers 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.

Design environments such as Framer [19], Janus [6, 8, 9], and Hydra [10] support reflection-in-action. Figure 1 shows facilities of the Janus family of design environments. The domain-oriented construction facility allows users to graphically visualize and manipulate a design. The construction analyzer critiquing facility critiques the design to give design feedback that is linked into a hypertext argumentation facility. The goal specification facility helps to keep critics relevant to the designer's objectives. Reflection-in-action is also supported by simulation facilities that allow what-if analysis as a further design evaluation.

Designers will gain the most from design feedback that is both timely and relevant to their current design task. Design environments can address timeliness by linking critics to a model of the design process. Framer uses a checklist to model the process of designing a user interface window. At any given time the designer is working on one checklist item and only critics relevant to that item are active. Design environments can also address relevance by

linking critics to specifications of design goals. For example, Janus and Hydra allow the designer to specify goals for kitchen floorplans, and thus activate only those critics relevant to stated design goals. Furthermore, Hydra uses critiquing perspectives, or modes, to activate critics relevant to a given set of design issues and deactivate irrelevant critics.

Existing design environments in the domain of software architecture emphasize automatic generation from formalisms instead of the cognitive needs of the software architects in making design decisions. The Aesop [11] system allows for a style-specific design environment to be generated from a specification of the style. The DaTE [2] system allows for construction of a running system from an architectural description and a set of reusable software components. AMPHION [15] is similar in that it allows users to enter a graphical specification from which the system can generate a running program. Each of these systems provides support for design representation, manipulation, transformation, and analysis, but none of them explicitly supports designers' cognitive processes. Argo can generate main procedures which combine software components into a running system. However, the main contribution of Argo to the software architecture community is its emphasis on the cognitive needs of the architect.

3. Design environment implementation

In this section we discuss our approach to implementing extensible design environments. First, we briefly describe the design representation that Argo uses internally. We follow that with a description of Argo's architecture. In describing our own design environment, we have attempted to present themes and details to aid other researchers in building their own design environments.

3.1. Internal representation

Argo stores designs internally as a connected graph with annotations on the nodes and arcs. Nodes represent design materials, while arcs represent relationships between those materials. Nodes and arcs are both first class objects with state and behavior. Annotations on nodes and arcs are also first class objects. In addition to the connectivity of the graph, annotations may be used to represent other important features in design perspectives, e.g., the location and size of design materials in a two-dimensional kitchen floorplan. Design materials are intelligent in that they may carry their own domain knowledge in the form of critics, simulation routines, or predefined transformations.

Rather than define a simple model of the design that selectively considers only a few design issues, we advocate including diverse design material types and relationships relevant to diverse design issues. This inclusive approach better suits the breadth of design issues that designers must consider.

The inclusion of diverse design issues can make for large, unwieldy connected graphs. We address this with design perspectives. Designs are manipulated through connected graph editors each of which shows a single design perspective, while annotations are edited via dialog boxes. The various perspectives are projections, or subgraphs, of the internal connected graph such that each perspective presents only materials and relationships relevant to a few design issues.

Since specific constraints on the design are handled in the critics, Argo's infrastructure makes very few assumptions about the domain-oriented characteristics of the graph. It is this simple, precise, and flexible design representation that allows Argo to separate domain-neutral code from domain-oriented code, present diverse design perspectives, and make use of first class supporting artifacts such as a process model. As will be seen in the next subsection, reusable domain-neutral facilities reside at the lowest level of the Argo virtual machine architecture, while the domain-specific facilities reside at higher levels.

In the domain of software architecture, nodes correspond to architectural elements such as conceptual software components, programming language modules, source code files, and operating system processes. Meanwhile, arcs represent relationships between those elements such as *implements*, *communicates-with*, and *is-allocated-to*.

3.2. Design environment architecture

Figure 2 presents facilities of our software architecture design environment. The designer uses multiple, coordinated design perspectives (Figure 3) to view and manipulate Argo's internal design representation. Automated design critics in the environment monitor the design and deliver design feedback when relevant and timely. Critics place their feedback in the designer's "to do" list. Each "to do" item reminds the designer to address an open design issue (Figure 4). The designer uses a process model as a resource in carrying out a design process, while the design environment uses that process model to ensure the timeliness of delivered knowledge (Figure 5).

For comparison, Figure 1 shows facilities of the Janus design environment. Like Janus, Argo provides a diverse set of facilities to support reflection-in-action including construction and critiquing mechanisms. Argo, however, extends these facilities by integrating them with a flexible process model and "to do" list that explicitly supports opportunistic design, and multiple, coordinated design perspectives to aid in comprehension and problem-solving. Each of these facilities and the cognitive theories that motivate them are discussed in the next section.

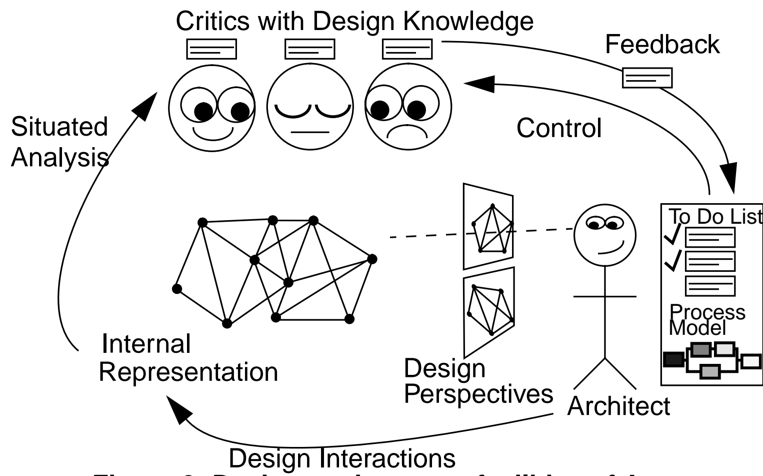

Figure 2: Design environment facilities of Argo

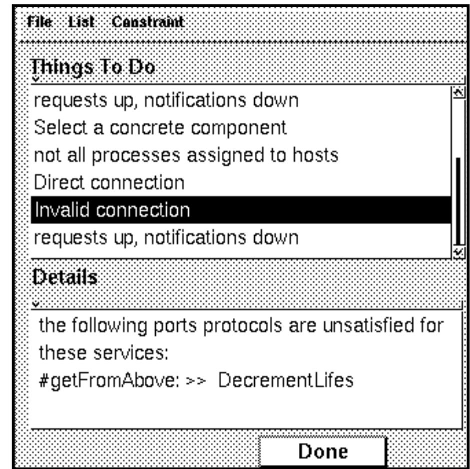

Figure 4: The architect's "to do" list

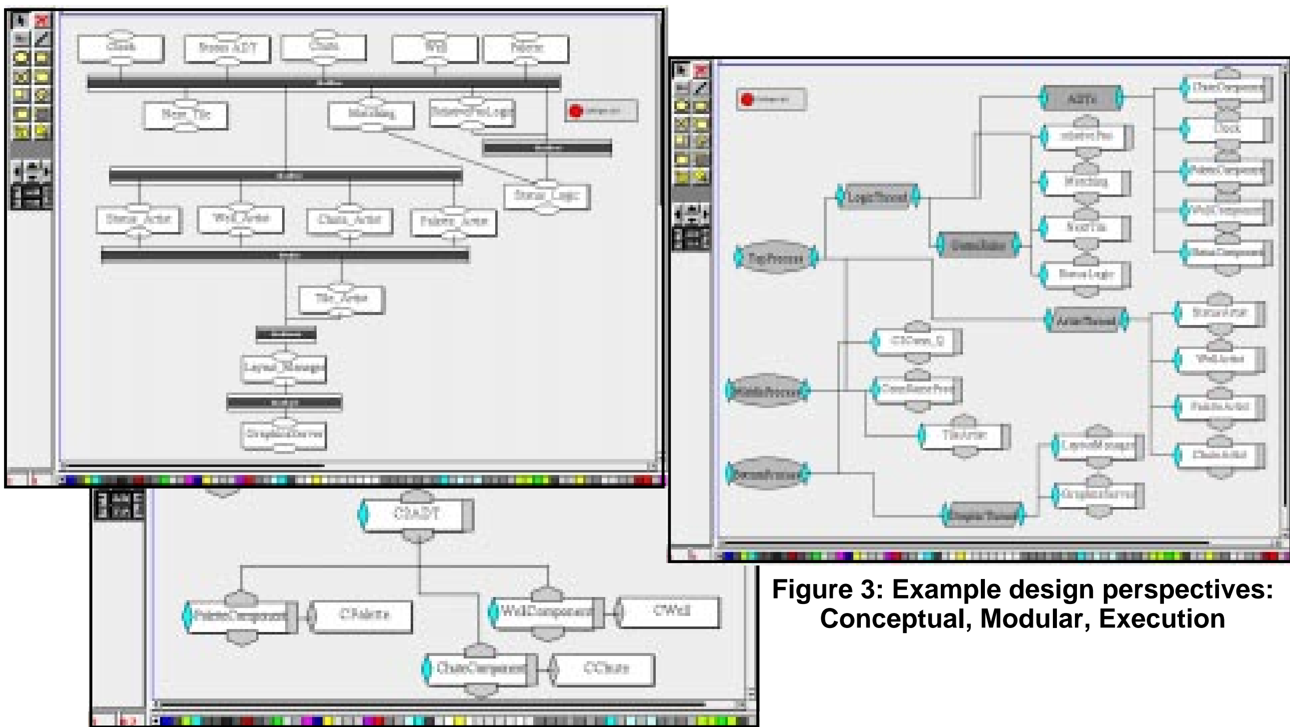
Figure 3: Example design perspectives: Conceptual, Modular, Execution

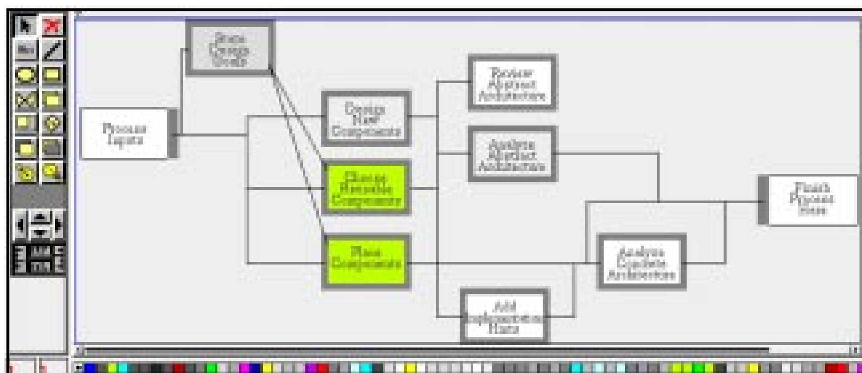
Figure 5: A software architecture design process perspective

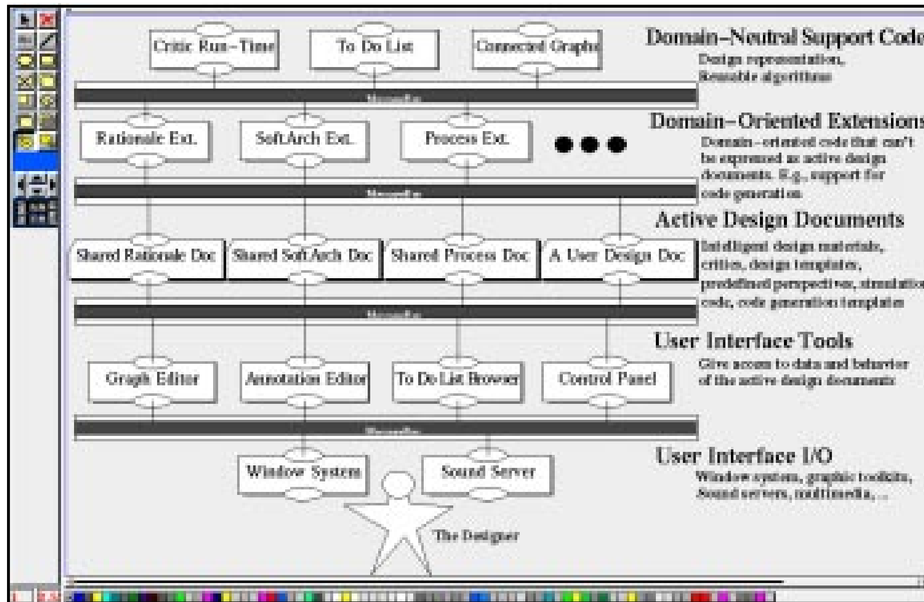

Figure 6: A screen shot of Argo modeling its own architecture in the C2 style

Figures 1 and 2 indicate what facilities are available to designers, but they give little indication of how the design environment is implemented. Janus and similar systems have tended to have one major software component for each facility. Those components form a knowledge-rich design environment with tight user interface, data, control, and process integration [16]. Our interest in software architecture motivated us to seek a more flexible and extensible architecture, while retaining a fairly high level of integration.

Figure 6 shows a screen shot of Argo modeling its own architecture in the C2 style [29]. The topmost row of software components provides domain-neutral support code. The second row allows multiple, independent, domain-oriented extensions. Each extension defines new facilities if needed, e.g., code generation support in the software architecture extension. The third row holds active design documents for the design being worked on. Design documents are active in that they contain intelligent materials with both state and behavior. Each extension also provides shared domain-oriented active documents containing a palette of intelligent materials, reusable design templates, and supporting artifacts. User design documents may reference (rather than include) code and data in the shared active documents. The fourth row contains user interfaces for designers to access the data and behavior of the active documents. The lowest row of components provides I/O needed to interact with the designer. In C2 style architectural models, 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 horizontal connector that broadcasts messages sent from one side to all components on the other side. Figure 7 presents the same architecture in a more traditional and less detailed virtual machine notation.

The first advantage of separating domain-oriented code from domain-neutral code is increased reusability across application domains. For example, the same critic run-time system could be used in domains such as software architecture or kitchen design. The second advantage is that within a given application domain, various first class supporting artifacts may be used together. Here “first class” means that they may be visualized, manipulated, and critiqued. For example, software architects can use the same modeling and critiquing facilities for design rationale and process modeling, provided that extensions for those domains are present.

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 users to interact with intelligent materials. The same trend can be observed in the general rise of object-oriented

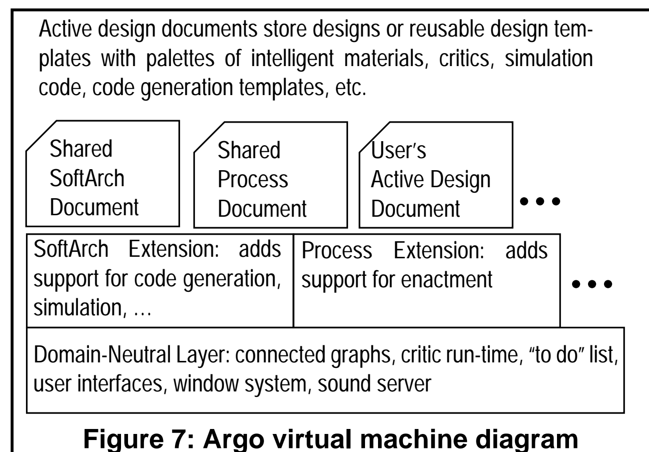

Figure 7: Argo virtual machine diagram

Table 1: Selected Argo architectural critics

Name of Critic	Critic Type	Decision Category	Explanation
Invalid Connection	correctness	checking	Mandatory message signatures not satisfied by adjacent components in the conceptual architecture
One Up One Down	correctness	checking	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	profiling	Calculated memory requirements exceed stated goals
Need more reuse	consistency	choosing	Percentage of reusable components is below stated goals
OS Incompatibility	consistency	annotating	Components have conflicting environmental requirements

and component-based approaches to software design. The advantages of this shift include increased scalability, better separation of concerns, and more opportunity for stakeholders with different skills to contribute to design environment seeding [7]. All of these are important in supporting the evolution of designs, design environments, and design communities over time.

4. New facilities for supporting cognitive needs

Our extensions to previous design environment facilities are motivated by theories of designers' cognitive needs. Designing a complex system is a cognitively challenging task, thus designers need cognitive support to create good designs. Specifically, we extend previous design environment facilities by enhancing support for reflection-in-action and adding support for cognitive needs identified in the theories of opportunistic design and comprehension and problem-solving.

4.1. Design feedback

Theory – reflection-in-action: As discussed in Section 2, Schoen's theory of reflection-in-action [23, 24] indicates that designers must iteratively construct designs, reflect on and revise each intermediate, partial design. Guindon, Krasner, and Curtis noted the same effect as part of one study of software developers [13]. 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.

However, there are inherent dangers in this “natural” evolutionary design process. It can allow artifacts to rapidly grow out of control: inconsistencies can evolve undetected, and some requirements may be overlooked while the designer focuses on more engaging ones.

Critics allow designers to follow the observed design process of reflection-in-action while retaining some of the positive properties of a rigorous software process. In particular, they augment a human designer's ability to consistently detect potential breakdowns, especially in the situations where designers are working with unfamiliar materials.

Table 2: Details of the Invalid Connection critic

Attribute	Value
Name	Invalid Connection
Type	Correctness
Decision Category	Checking
Smalltalk Predicate	`[:comp | | invalidServices | invalidServices := comp inputs , comp outputs select:[:s | s isSatisfied not]. invalidServices isEmpty not. ]`
Hushed	False
Feedback	“The following port protocols are unsatisfied for these services:” <<a list of ports and services>>
Expert	jrobbins@ics.uci.edu

Implementation in Argo: Argo's infrastructure contains specific features to support critics. First, Argo has a framework for implementing critics and a run-time facility for evaluating the predicates of active critics. Predicates are currently implemented as a combination of code fragments and cross-reference tags for critic type and decision category. Second, Argo has a variety of feedback control mechanisms for controlling which critics are active at a given time and for managing design feedback. Some examples of critics in the domain of software architecture are given in Table 1, while Table 2 presents one critic in detail.

Argo associates critics with intelligent design materials; there is no central rule base of critics. When a new type of design material is defined, new critics may be defined for it. Critics which cannot easily be associated with any one design element may be associated with one or more design perspectives. For simplicity, Figure 2 presents critics as looking down on the design from above. A more literal presentation would show critics associated with each node of the design representation, looking around at their neighbors. Our current Argo prototype has less than 20 critics, but future research prototypes or deployed design

environments could have hundreds or thousands of critics. Dividing critics among materials helps to make Argo more scalable by loading only critics that could be useful in the design at hand.

In the field of software architecture, several authors have identified the need for architectural design guidance [1, 12, 29]. One approach to representing that knowledge is the compilation of architectural styles. Architectural styles provide design guidance by suggesting constraints on design decisions and ways to factor design complexity. Styles are based on a set of recurring patterns observed in a given domain. Styles may also guide the design environment builder, in that critics and perspectives may be organized according to style.

While some of the assumptions of software components, connectors, or styles are implicit, it is usually possible to make them explicit as rules, even if merely as rules of thumb. We anticipate that much of the practical, day-to-day knowledge about software architectures will take the form of guidelines or rules of thumb, and styles will accumulate so many of them that automated support will be needed. English grammar checking tools are in an analogous situation: some rules are too complex or fuzzy to implement precisely, and a typical critique produces enough feedback to overwhelm the user.

We address this issue in Argo with feedback management techniques. These techniques ensure that criticism is timely and relevant, and also help designers make sense of criticism by organizing it. Critics may be active or inactive depending on the state of the design, the design process, and stated design goals. A control panel allows the designer to deactivate groups of critics by type. If individual critics are providing inappropriate feedback or are felt to be too intrusive, the designer may hush them, rendering them temporarily disabled. Below we describe two more powerful feedback control mechanisms in more detail: associating critics with steps in a design process and the "to do" list user interface.

4.2. Design process support

Theory – opportunistic design: The cognitive theory of opportunistic design tells us that designers deviate from plans, even their own plans, in order to minimize the cognitive cost of context switches between design tasks [20, 27, 28, 30]. These deviations may be desirable from a cognitive perspective, but they lead designers into a variety of difficulties as discussed in the Guindon, Krasner, and Curtis study [13].

We extend previous design environment work by maintaining a model of the design process, and using it to accommodate opportunistic design. Design environments can address the cognitive needs of designers by focusing on certain design process characteristics: flexibility, visibility, reminding, delayed commitment to detail, and timeliness of feedback.

Flexibility is the foremost design process characteristic. Designers must be allowed to deviate from a prescribed sequence and allowed to choose which goal or problem is most effective for them to work on. Designers must be able to add new goals or otherwise alter the design process as their understanding of the design situation improves. The process model serves primarily as a resource to the designer's cognitive process, and only secondarily as a constraint on it.

Visibility helps designers orient themselves in the process, thus supporting opportunistic choice of design tasks. In particular, the design process model should be able to represent what has been done so far and what is possible to do next. Furthermore, visibility enables designers to take a series of excursions into the design space and re-orient themselves afterwards to continue the design process.

Reminding helps architects revisit incomplete details or overlooked alternatives. Reminding is most needed when design alternatives are many and when design processes are complex or driven by exceptions.

Delayed commitment to details is supported by the combination of flexibility, visibility, and reminding. If architects are forced to fully formalize each tentative design decision prior to doing any analysis, then the effort required to explore design alternatives will be quite high. Higher effort means that fewer alternatives will be considered, which reduces confidence in the design. Higher effort also shifts attention away from the design at hand and into planning required to use the tool efficiently. In our approach, the ability of critics to deliver partial critiques of partial designs in a managed and usable form allows designers to evaluate their designs without premature commitment to design details.

Timeliness of feedback is the final design process characteristic. Critics deliver information to aid designers in decision making. To produce timely feedback, critics must have an explicit model of the design process and the designer's progress in it. When the design process is modeled as tasks that each address only a few design issues, then knowing which tasks are in progress indicates which decisions are being considered, and thus which critics are timely. Criticism is distracting when it involves design decisions that the architect has not yet begun considering. The designer can also indicate when a task is considered finished, and the design environment can generate additional criticism at that time, perhaps marking the task as still in progress if there are high priority "to do" list items pending. In addition to improving design decisions, timely feedback helps the designer make timely process decisions, e.g., "is this design excursion complete?, "does a past

decision need reconsideration?", "what decisions might be considered next?"

Implementation in Argo: Motivated by the theory of opportunistic design, we have attempted to move from predefined processes that force a certain design decision ordering to flexible process models that allow the architect to minimize cognitive costs of context switches. We extend previous work in design environments by introducing an explicit, first class model of the design process with progress information, and a more flexible "to do" list user interface for presenting design feedback. The "to do" list suggests, but does not force, a particular decision ordering. The process model also supports reflection-in-action by helping to keep design feedback timely.

Argo's process modeling extension uses an IDEF0-like notation to model the tasks involved in a typical design process (Figure 5). Each task in the design process works on input produced by upstream tasks and produces output for consumption by downstream tasks. No control model is mandated: tasks can be done in any order (provided needed inputs are available), tasks can be repeated, and any number of tasks can be in progress at a given moment. Each task is marked with a status: future, in progress, or done. Status information is shown graphically via color in the process diagram. Each task is also marked with a decision category symbol: building, choosing, checking, annotating, or profiling. Decision category symbols and statuses are used to limit the activity of critics and thus avoid producing feedback that is not timely and relevant.

The design process model shown in Figure 5 is a fairly simple one, partly because the C2 architectural style does not impose any explicit process constraints, and partly because this example does not consider issues of organizational policy. In practice, the process would be more complex. The process of defining and evolving the process (usually called the meta-process) is itself a complex, evolutionary task for which process designers may need feedback and other design environment facilities.

The process model in Argo is first class. It is represented as a connected graph of intelligent design materials. Multiple perspectives may be defined to view the process as appropriate for various stakeholders. The designer may define, modify, and annotate the process model via the same editor used to work on architectures. Critics may operate on the process model to check that it is a "good" process and guide its construction and modification. For instance, one simple process completeness critic carried by task nodes checks that "the output of this task should be used by some other task." The same techniques that are used to control architecture critics can be used to manage process critics. Those techniques include modeling the meta-process and the designer's progress in it, so that process critics will be relevant and timely. While the ability to change the process gives freedom to individual designers, critics provide a mechanism to communicate or enforce externally imposed process constraints.

The "to do" list user interface presents design feedback to the designer (Figure 4). Most "to do" items are posted by critics, however items may come from process enactment, or the designer himself. When the designer selects a pending feedback item from the upper pane, the associated (or "offending") design materials are highlighted in all perspectives and details about the open design issue and possible resolutions are displayed in the lower pane.

The priority of items on the designer's "to do" list is estimated based on the predefined severity of the criticism and the state of the process model. Designers may address issues in an order they choose, although items are sorted by estimated priority. Designers may also reorder the list or insert items as reminders to themselves. The "to do" list complements the process model in providing flexibility, visibility, and reminding.

4.3. Design perspectives

Theory – comprehension and problem-solving: The theory of comprehension and problem-solving observes that designers benefit from seeing their designs from different design perspectives [5, 14, 17]. This is true, in part, because the availability of multiple perspectives increases the chance that the designer will see a simple mapping between one of them and his or her mental model of the problem being addressed. Coordination among the design perspectives means that materials and relationships presented in multiple perspectives may be viewed and manipulated in any of those perspectives. Overlapping, coordinated perspectives aid understanding of new perspectives, and thus new design issues, by allowing the designer to apply knowledge of one perspective to another [18].

Dividing the complexity of the design into multiple perspectives allows each perspective to be simpler than the overall design. Moreover, separating concerns into separate perspectives allows information relevant to certain related issues to be presented together in an appropriate notation.

Good designs usually have organizational structures that allow designers to locate design details. However, in complex designs the expectation of a single unifying structure is a naive one. Complex software system development is driven by a multitude of forces: human stakeholders in the process and product, functional and non-functional requirements, and low-level implementation constraints.

It is our contention that no fixed set of perspectives is appropriate for every possible design; instead perspective views should emphasize what is currently important in the project. When a new set of issues arises in the design, it may

be appropriate to use a new perspective on the design in addressing those issues. While we emphasize the evolutionary character of design perspectives, useful sets of initial perspectives can often be identified ahead of time in specific domains.

Implementation in Argo: Figure 3 shows three perspectives produced with Argo. The system shown is a simple video game called KLAX where falling colored tiles must be arranged in rows and columns. In the *conceptual architecture*, small rectangles represent software components and connectors, while arcs represent communication pathways. Small ovals on the components represent the communication ports of each component. The *execution architecture* hierarchically groups modules into operating system processes and threads. The *modular architecture* maps conceptual components to programming language modules. These perspectives provide much of the information needed for system generation, assuming the component themselves are already written.

Argo supports multiple, coordinated perspectives with customization [21]. In addition to the views described in this paper, Argo allows for the construction of new perspectives and their integration with existing perspectives. As noted in [26], architects who are given a fixed set of formal notations often revert to informal drawings when those notations are not applicable. One goal of Argo is to allow for the evolution of new notations as new needs are recognized. In addition to the structured graphics representing the architecture and process, Argo allows designers to annotate perspectives with arbitrary, unstructured graphics (as demonstrated in Figure 6). Customizable presentation graphics are needed because the unifying structures of the system under construction must be communicated convincingly to other designers and system implementors. To be convincing, the style of presentation must fit the professional norms of the development organization: it should look like a presentation, not a designer's scratch pad. Furthermore, ad-hoc annotations that are found to be useful can be incrementally formalized and incorporated into the notations of future designs [25]. We expect that Argo's low barrier to customization will encourage evolution from unstructured notations to structured ones as recurring formalization needs are identified.

Soni, Nord, and Hofmeister [26] identify four architectural views: (1) conceptual software architecture describes major design elements and their relationships; (2) modular architecture describes the decomposition of the system into programming language modules; (3) execution architecture describes the dynamic structure of the system; and (4) code architecture describes the way that source code and other artifacts are organized in the development environment. Their experience indicates that separating the concerns of each view leads to an overall architecture which is more understandable and reusable. In demonstrating Argo, we chose several of the same perspectives; however, we believe that the choice of perspectives depends on the type of software being built and the tasks and concerns of design stakeholders.

5. Conclusions and future work

In this paper we have presented the architecture and facilities of Argo, our software architecture design environment. Argo's architecture is motivated by the desire for reuse and extensibility. Argo's facilities are motivated by the observed cognitive needs of designers. The architecture separates domain-neutral code from domain-oriented code and intelligent design materials. The facilities extend previous work in design environments to support reflection-in-action, opportunistic design, and comprehension and problem-solving.

In future work we will continue the themes of our current research. Further identification of the cognitive needs of designers will lead to new design environment facilities to support those needs. Also, we will seek ways to better support the needs that we have identified in this paper, e.g., a process model that explicitly represents the cognitive cost of switching design tasks. Customizability of design environments is an important cross-cutting issue that we will exploit further.

Our current prototype of Argo is robust enough for experimental usage. In fact, we are using it to design the next version. However, 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/edcs.

Acknowledgments

The authors would like to acknowledge Richard Taylor (UCI), Gerhard Fischer (CU Boulder), David Morley (Rockwell International), and Peyman Oreizy, Nenad Medvidovic, and other members of the Chiron research team at UCI.

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. 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.
3. Engelbart, D. A Conceptual Framework for the Augmentation of Man's Intellect. In: Greif I, ed. *Computer-Supported Cooperative Work: A Book of Readings*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1988, 35-66.

4. Engelbart, D. Toward Augmenting the Human Intellect and Boosting our Collective IQ. *Communications of the ACM* 1995, vol.38, no.8, 30-33.
5. Fischer, G. Cognitive View of Reuse and Redesign. *IEEE Software*, Special Issue on Reusability 1987, vol.4, no.4, 60-72.
6. Fischer, G. Domain-Oriented Design Environments. *Proc. of The 7th Knowledge-Based Software Engineering Conference,* 204-213.
7. Fischer, G., Girgensohn, A., Nakakoji, K., and Redmiles, D. Supporting Software Designers with Integrated Domain-Oriented Design Environments. *IEEE Transactions on Software Engineering*, June 1992, vol.18, no.6, 511-22.
8. 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.
9. Fischer, G., Lemke, A., McCall, R., and Morch, A. Making Argumentation Serve Design. *Human-Computer Interactions*, 1991, vol.6, no.3-4, 393-419.
10. 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.
11. 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, December 1994, vol.19, no.5, 175-88.
12. Garlan, D., Allen, R., and Ockerbloom, J. Architectural Mismatch: or Why it's hard to build systems out of existing parts. *International Conference on Software Engineering 17*, 1995, 179-185.
13. 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: Ablex Publishing Corporation, Lawrence Erlbaum Associates, 1987, 65-82.
14. Kintsch, W. and Greeno, J. G. Understanding and Solving Word Arithmetic Problems. *Psychological Review,* 1985, vol.92, 109-129.
15. Lowry, M., Philpot, A., Pressburger, T., Underwood, I. A Formal Approach to Domain-Oriented Software Design Environments, KBSE'94, 48-57.
16. Thomas, I. and Nejmeh, B. Definitions of Tool Integration for Environments. *IEEE Software*, March 1992, vol.9, no.2, 29-35.
17. Pennington, N. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, vol.19, 1987, 295-341.
18. Redmiles, D. F. Reducing the Variability of Programmers' Performance Through Explained Examples. *INTERCHI '93 Conference Proceedings*, April 1993, 67-73.
19. Rettig, M. Cooperative Software. *Communications of the ACM*, April 1993, vol.36, no.4. 23-28.
20. Rist, R. Variability in program design: the interaction of knowledge and process. *The International Journal of Man-Machine Studies*, 1990, 1-72.
21. 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.
22. Robbins, J. E. and Redmiles, D. F. Software Architecture from the Perspective of Human Cognitive Needs. *Proc. of the California Software Symposium (CSS'96)*, April 1996, 16-27.
23. Schoen, D. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books, 1983.
24. Schoen, D. Designing as Reflective Conversation with the Materials of a Design Situation. *Knowledge-Based Systems,* 1992, vol.5, no.1, 3-14.
25. Shipman, F. and McCall, R. "Supporting Knowledge-Base Evolution with Incremental Formalization," *Human Factors in Computing Systems, CHI'94 Conference Proceedings*, Boston, MA, 1994, 285-291.
26. Soni, D., Nord, R., and Hofmeister C. Software Architecture in Industrial Applications. *International Conference on Software Engineering 17*, 1995, 196-207.
27. Soloway, E. and Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering,* 1984, vol.10, no.5, 595-609.
28. 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.
29. 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, vol.22, no.6, 390-406.
30. Visser, W. More or Less Following a Plan During Design: Opportunistic Deviations in Specification. *International Journal of Man-Machine Studies*, 1990, 247-278.