Design Rationale Systems: Understanding the Issues

Jintae Lee, University of Hawaii

MOST CURRENT DESIGN RATIONALE SYSTEMS FAIL TO CONSIDER PRACTICAL CONCERNS, SUCH AS COST-EFFECTIVE USE AND SMOOTH INTEGRATION. THE AUTHOR IDENTIFIES SEVEN TECHNICAL AND BUSINESS ISSUES AND DESCRIBES THEIR IMPLICATIONS.

IN THE LAST FEW YEARS, INTEREST in design rationales has grown. Design rationales are important tools because they can include not only the reasons behind a design decision but also the justification for it, the other alternatives considered, the tradeoffs evaluated, and the argumentation that led to the decision. The use of a design rationale system—a tool for capturing and making design rationales easily accessible—can thus improve dependency management, collaboration, reuse, maintenance, learning, and documentation. However, if such systems are to keep pace with the growing and changing demands of design technology, researchers and developers must begin to answer certain questions.

In this article I identify seven issues, which I have derived from an informal (undocumented) survey of major existing design rationale systems and discussions with workshop participants, including those in the 1992 AAAI Design Rationale Capture and Use Workshop.[1] The issues identified include what services to provide; what parts of the rationale to represent explicitly; how to represent, produce, and access rationales and manage them cost effectively; and how to integrate the design rationale system.

My goal in writing this article is to help researchers and developers of future design rationale systems understand the available options and tradeoffs. No one system can hope to address all these issues. Some will emphasize one thing; others, another. Indeed, these issues delineate the major dimensions along which a design rationale system is likely to differ. But if the community can better understand each of these issues, it will be more equipped to produce design rationale systems that succeed in their particular application areas.

What services to provide

The services a design rationale system provides will determine almost all other aspects of its design, such as what to represent and how to represent the rationales.

Figure 1 shows common services classified into four major groups according to the user group who benefits: better design (designers), better maintenance (system maintainers), learning (new trainee, students, learning programs), and documentation (future designers and maintainers). The arrows between services indicate that a service at the end of the arrow (tail) supports the service at the beginning (arrowhead). For example, using design rationales to support dependency management or problem solving or simulation and diagnosis improves design and maintenance, and supports learning. Using design rationales to support project management helps support collaboration, requirements engineering, or reuse—any one of these in turn can result in better design.

Better design support. Well-structured design rationales can help designers track the issues and alternatives being explored and their evaluations. This, in turn, clarifies the overall structure of the reasoning process and supports decision making. In some cases, of course, all this information can actually hinder design activities by imposing unnecessary overhead or breaking the natural flow

0885-9000/97/$10.00 © 1997 IEEE

of design activities, but in most cases this clarification leads to better design.

Theoretically, one design rationale system can support all the services described here both during design and before—for example, during requirements definition.

Dependency management. Design can be viewed as the process of managing dependencies to yield a product that honors all dependencies among requirements and the components that implement them. Design rationales can make explicit dependency relations among design parts, decisions, arguments, and alternatives so that they work together consistently. Some systems provide dependency management simply by displaying issues that depend on the current issue,[2] while more complex systems actually detect conflicts among various constraints.[3]

Collaboration/project management. Design rationales also provide a common foundation when multiple parties are involved. Explicitly represented rationales can provide common vocabulary and project memories, and make it easier to negotiate and reach consensus. In Shared-DRIM,[3] for example, whenever a design agent makes a recommendation, the system checks to see if an existing recommendation might conflict with it. If so, the system informs all relevant parties about the change made to the object, identifies the cause of the change, and looks for a constraint violation. In this way, interactions among, say, engineers and contractors are reduced, which helps design proceed more efficiently.

Design rationales also help designers or computational agents track unresolved issues and their dependencies, train newcomers, restore interrupted designs, and support distributed task allocation.

Reuse/redesign/extension support. Design rationales help reuse in two ways. First, they can serve as indices to past knowledge (similar designs, parts, problems encountered). SoftDA,[4] for example, supports reuse by acquiring the relationship information about designs and requirements and using it to index documents and codes.

Designers can also reuse the rationales themselves. For example, the precedent management in Sibyl[5] uses the goals from past decision rationales to suggest potentially relevant alternatives, and uses both goals and alternatives to retrieve potentially relevant arguments evaluating the alternatives.

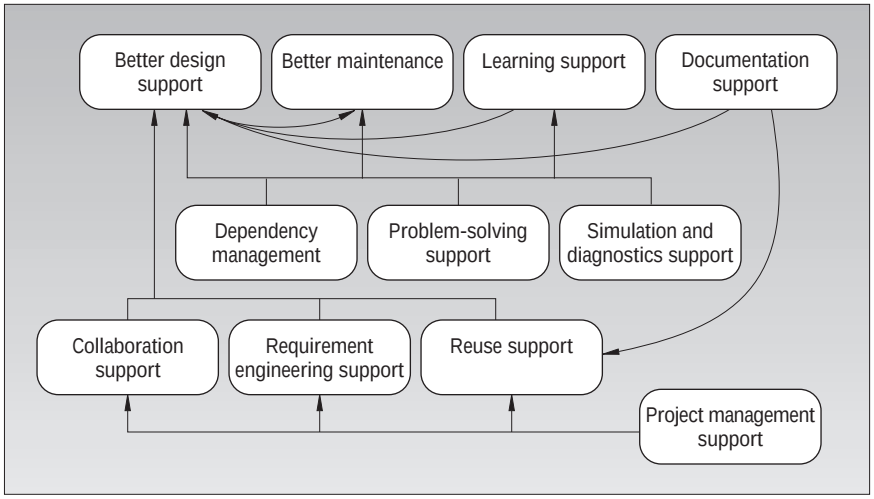

Figure 1. Services provided by most design rationale systems. Arrows between services indicate support relations.

For reuse to be successful *and* cost-effective, however, the design rationale system must also make it cost-effective to sort out knowledge that is outdated or context-sensitive. Without this provision, its suggestions must be taken with a large grain of salt. Mary Lou Maher and Andres Gomez de Silva Garza discuss other problems in the reuse of design knowledge.[6]

Better maintenance support. Because design rationales explain the design decisions made, they can also help maintain the design. Most existing systems provide this service in its simplest form: comments documenting program codes. EES,[7] on the other hand, extracts much richer development rationales and uses them to generate more sophisticated explanations for system maintenance.

Learning support. Design rationales can help both people and systems learn mutually and interactively. Janus,[8] for example, provides computational agents, or *critics,* which have access to design rationales and monitor human designers. When a critic encounters a human decision that is suboptimal according to its knowledge, it presents the designer with an appropriate recommendation along with its rationales. The designer can either agree and accept them, thus acquiring a new piece of knowledge, or disagree and teach the system by supplying a new piece of knowledge so that the critic knows better next time.

Of course, many learning systems attempt to pick up knowledge from past problem-solving traces. And if you view a problem-solving process as the process of designing a solution, technically all these systems can be design rationale systems, especially those that learn by analogy (see articles by Ashok Goel and Alex Duffy in this issue).

Documentation support. Design rationales can be used to automatically generate documentation. I mention documentation as a category apart from better design because documents help others besides designers. Managers or users can use them to evaluate the design. Lawyers can use them to determine if the design is intellectual property. In fact, Frank Shipman and Ray McCall,[8] who articulate three main perspectives of design rationales—argumentation, communication, and documentation—argue that the documentation perspective has been the most successful and should provide a model for the design of design rationale systems (see "How to access rationales").

What to represent explicitly

It is impossible to represent an entire design rationale explicitly. Rationales are embedded not only in formal documents such as design specifications, meeting abstracts, and interface documents, but also in informal media such as phone conversations, blackboard sketches, and discussions over lunch. In fact, almost anything in a design process may be a part of a design rationale as long as it is represented and can be used to trace a reason for some aspect of the design. Whatever is represented, however, must be accessible, so it must have some structure. This is why, say, videotaping all designer interaction would not be suitable. The unstructured nature of the recording would make it difficult to access exactly what is needed and use that information in any quantitative way.

Functional dependency. As is true for any representation, what you represent depends on what you want to do with it. Shipman and McCall[8] note that if design rationales are used primarily to ensure careful reasoning and better problem solving among the designers (argumentation perspective), then the logical structure of the reasoning must be made explicit so that the system can do the logical bookkeeping and detect any potential errors. On the other hand, if the rationales are used primarily to enable outsiders to understand or regulate design activities (documentation perspective), then only the results of the reasoning and their immediate explanations must be documented—not all the possibilities and the dead ends explored. Finally, if rationales are used to support project management, then constructs and attributes that reflect project status must be provided, such as unresolved or pending issues and deadlines and the people responsible for them.

Generic structure. In my survey of existing and proposed design rationale systems and in my discussions with workshop participants, I was able to find a generic structure to what was being represented—despite major differences in system implementation. This structure takes the form of layers, with some layers specializing other layers or requiring the presence of others. I identified three major layers: decision, design artifact, and design intent.

Decision layer. The decision layer characterizes the generic structure of a decision process, regardless of use. It comprises five sublayers: issue, argument, alternative, evaluation, and criteria. Figure 2 shows how providing additional constructs progressively for each sublayer can differentiate the components of design rationales and enrich their representation.[5] The argumentation layer (Figure 2a) makes explicit the arguments underlying a decision and their relations (supports, refutes, qualifies). The alternative layer (Figure 2b) makes explicit individual alternatives and their relations (component-of, incompatible, specializes). With constructs for this layer (and the constructs linking them to those in the argument layer), the system can sort an undifferentiated body of arguments and associate them with individual alternatives.

The evaluation layer (Figure 2c) makes explicit the evaluation measure and the relations used (nominal, ordinal, real values, maximum expected utility). With the evaluation layer, the individual evaluations become accessible to the system, which might use them to rank the alternatives or recompute the evaluations when related evaluations change. The criteria layer (Figure 2d) makes explicit the criteria used and their relations (mutually exclusive, tradeoffs, specializes). With constructs for this layer, the system can group evaluations and the arguments by alternative and criterion, display them (in a table, for example), and update evaluations when the relevant criteria change. The issue layer (Figure 2e) makes explicit the individual issues and their relations (generates, depends-on, replaces). With the issue layer, the system can now track all issues that depend on a particular issue, all issues that are yet to be resolved, issues that replace a given issue, and so on.

SYSTEMS ALSO DIFFER IN THE RICHNESS OF THE CONSTRUCTS THEY PROVIDE TO REPRESENT A PARTICULAR SPACE. GENERALLY, THE LESS A SYSTEM REPRESENTS, THE LESS OVERHEAD IT WILL IMPOSE—BUT THE TRADEOFF IS FEWER SERVICES.

The constructs underlying these sublayers appear under different names and not all are always present explicitly. However, even systems that do not make them explicit represent them implicitly as text. For example, gIBIS[2] explicitly represents only the decision layer, calling it "issue," the alternative layer (position), and the argument layer (arguments). DRL[5] provides similar constructs for the sublayers, calling them decision problem, alternative, and claim, respectively, and adds constructs for the criteria layer (criteria). Of course, a design rationale system must provide constructs for a layer before it can see the designer's actions. Hence, if the system supports comparing alternatives along different criteria, it must also provide constructs for representing the alternative and the criteria spaces.

Systems also differ in the richness of the constructs they provide to represent a particular space. For example, the constructs in DRCS[9] for the evaluation space let designers characterize the evaluation of alternative design specifications, reveal how well specifications have been achieved (has-importance, is-more-important-than, has-subattribute, has-subspecification, specification, version, achieves), and provide evaluation history (version, achieves relation between version and specification). However, these services come at the cost of high overhead. Generally, the less a system represents, the less overhead it will impose—but the tradeoff is fewer services.

Design artifact layer. If a design activity is viewed solely as a set of decision-making steps, the rich information about how the design components are related and how the information relates to individual decisions will remain implicit, if it is there at all. The design artifact layer takes a broader view, making explicit both decision-making steps and the related information.

A system can also provide constructs for representing design artifacts but not decisions. For example, FR[10] expresses only the functional or causal relation among design components and requirements. Again, the system's intended function explains this representation. The author notes that the design rationales can be used for tasks such as simulation, verification, and diagnosis, but the system would not be able to answer questions about the alternatives and arguments explored.

Systems also differ in the richness of the constructs they provide for this layer. Colin Potts and Glen Bruns[11] designed a system that explicitly introduces a construct in this space (artifact) and relates it to the decision layer. The artifact-synthesis layer in DRCS provides a richer set of constructs such as module, interface, connection, and constraints for representing artifacts and related operations.

Although a design rationale system should not be expected to represent or reason about all domain-specific design artifacts, it should at least provide a way to link design rationale objects (such as those in the decision layer) to relevant design artifacts. For example, a system need not provide a construct such as Hydraulic Jack, but it should provide a construct, say Artifact, that it knows can be decomposed into other artifacts, each of which can be associated with an issue or a set of constraints.

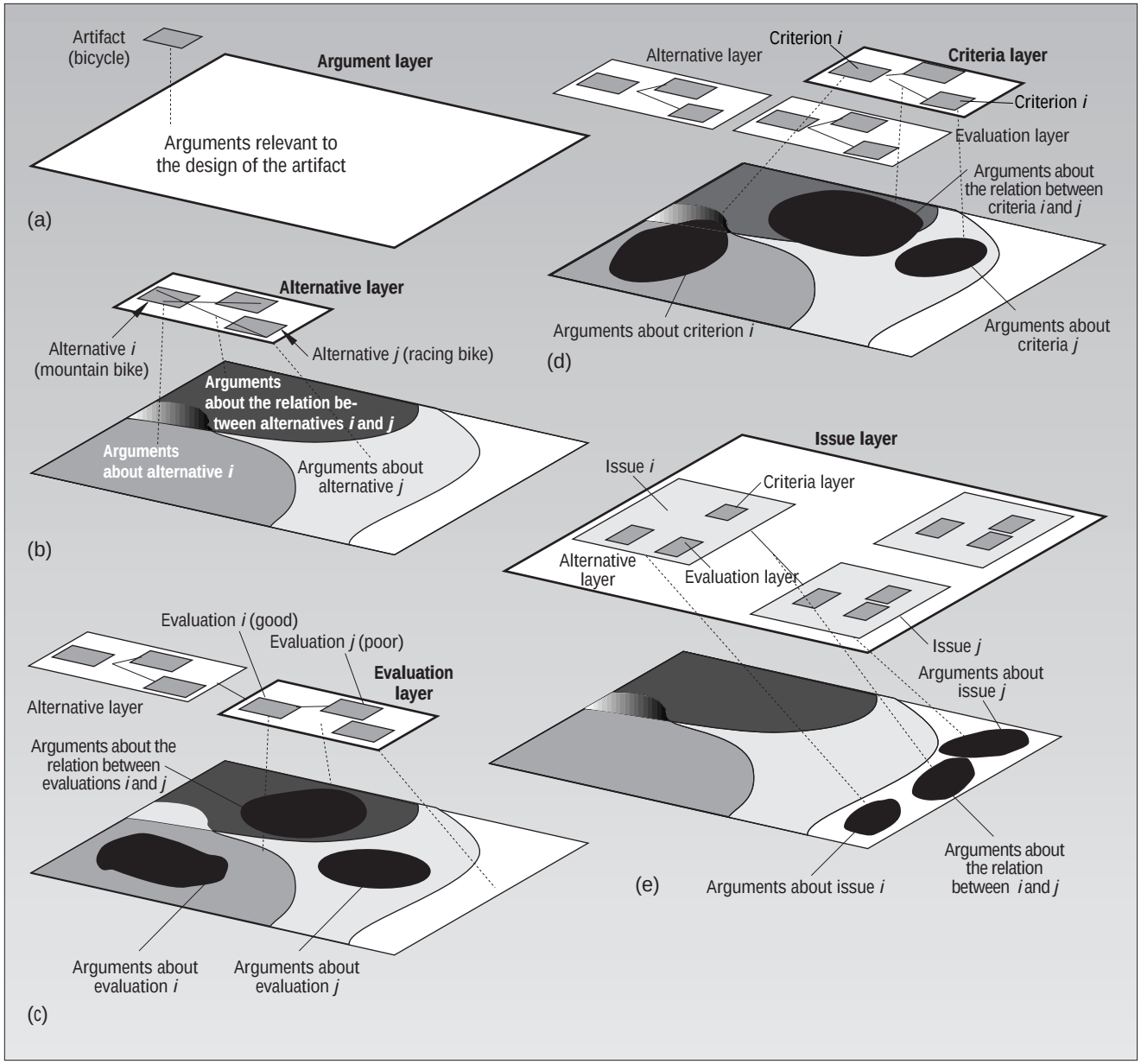

Figure 2. A progressively differentiated representation of design rationales within each of five sublayers in the decision layer.

Design intent layer. The third layer represents the metainformation underlying design decisions, such as intents, strategies, goals, and requirements. Once represented, this layer allows a design rationale system to reason about the goal, intent, or plan responsible for a given design. For example, from goals the system can derive criteria for evaluating alternatives so that when goals change, it can update or at least flag the criteria as outdated. The constructs provided for this layer range from simple constructs, such as objective and goal to complex constructs that capture relations, such as is-more-important and has-higher-priority.

How to represent rationales

The details of how design rationales are to be represented will depend on the system's representation language. However, the degree of formality is a more generic issue. For the sake of discussion, I assume a representation can be informal, semiformal, or formal, although formality is typically a continuum, not a set of categories with thresholds.

Once again, the choice of approach depends largely on the services the system will provide. Informal representation captures rationales in an unstructured form: descriptions in a natural language, audio/video recordings, and raw drawings. Informal descriptions are easy to create, but the system cannot interpret them, making them ill-suited for most computational services. The Electronic Notebook, developed by Fred Lakin of the Performing Graphics Company, attempts to overcome this limitation by using parsing technology. However, the technology is not yet mature enough to produce reliable interpretations of complex descriptions.

A semiformal representation is best if the primary services are to help people archive, retrieve, and examine the reasons for their decisions. In a semiformal representation, only parts of the representation are computer

readable; the rest is informal. In the systems with a semiformal representation, such as Sibyl,[5] the user interacts typically with different types of templates by filling out their attributes, either by typing in a natural language or by selecting from a menu of options. A semiformal representation can support many computational services not possible with an informal representation and may even require less overhead in capturing rationales because the system can suggest what information is expected (typed forms and options). However, the set of computational services would depend on how much of the representation was formal.

In a formal representation, objects and relations are defined as formal objects that the system can interpret and manipulate using formal operations. The creation of design rationales thus becomes a matter of creating a knowledge base in some formal language. The kind of formal representation will depend on the operations to be performed (deductive inference, associate defaults, inheritance). It can be any one of several standard representations: rules, frames, and logic. An example is any machine-learning system that uses its past problem-solving traces for future problem solving.

The more formally rationales are represented, the more services the system can provide. However, formalizing knowledge is costly. One way to reduce cost is to formalize it incrementally—essentially transforming a semiformal representation to a formal one. Thus, rationales can be captured with less overhead (because they are captured in a semiformal state), but once formalized can be used to support more computational services.

uiSibyl,[12] for example, starts with an informal requirements description, extracts relevant keywords, and determines if it can replace or describe any keywords with existing formal objects. If not, it attempts to help formalize the keywords by offering potentially related formal objects. Gerhard Fischer and Kumiyo Nakakoji of the University of Colorado attempt to incrementally formalize rules or domain objects from informal texts, using keywords.[13] mSibyl, proposed by Lukas Reucker and Warren Seering of the Massachusetts Institute of Technology, offers still another alternative: The user supplies a restricted form of English sentence as an attribute value of a semiformal object such as Decision and the system parses it into a formal language. Although the system's attempt to formalize hardly succeeds the first time, the incremental formalization approach is still appealing because it changes creation to the easier process of reaction and modification.

How to produce rationales

Design rationales can be produced in many ways, with the system participating entirely alone, somewhat, or not at all. The approaches described here are in order of minimum to maximum system participation.

Reconstruction. In this approach, people produce design rationales without using the system. They can do so by reasoning from their existing knowledge (introspection), conducting interviews with those involved in the design, or capturing rationales in raw form, such as video, and then translating them into a more structured form. Yet another way is to reverse-engineer a design by trying to infer a plan from the design artifact itself through a general knowledge of functions and device behaviors.

FORMALIZING KNOWLEDGE IS COSTLY. ONE WAY TO REDUCE COST IS TO FORMALIZE IT INCREMENTALLY—ESSENTIALLY TRANSFORMING A SEMIFORMAL REPRESENTATION TO A FORMAL ONE.

Reconstruction allows more careful reflection on the representation and layout of the rationales and does not disrupt the designers' activities. Jeff Conklin and K. Burgess-Yakemovic[2] report on using gIBIS at NCR, saying that the reconstruction process helped them identify several design omissions that would have cost three to six times more than the cost of capturing and reconstructing the rationales. Shipman and McCall[8] claim that the success of the documentation perspective of design rationales (as opposed to the communication and argumentation perspectives) owes much to the post-hoc creation of documentation. On the down side, the cost of reconstruction is high and it may introduce the biases of the person producing the rationales.

Record-and-replay. In this approach, rationales are captured as they unfold—for example, as designers use a shared database to raise issues, propose alternatives and criteria, and enter evaluations. The rationales can be captured synchronously (via videoconferencing or a shared screen that participants use to interact with one another) or asynchronously (via bulletin board or e-mail-based discussion). Systems that have informal and semiformal representations tend to use this approach because a representation that is too rich or too formal would create excessive overhead and disrupt the flow of design activities.

Methodological byproduct. In this approach, design rationales naturally emerge from the process of designers following a certain method. For example, in EES, the developers of an expert system follow a specific method that EES supports. The method's steps are in essence different kinds of refinements and reformulations, which the system captures and uses to generate explanations. Another example is the transformational approach,[14] in which a design rationale is defined as a trace of the decomposition of the heuristic methods into submethods and finally into applied transformations. The designer uses the transformations to verify that the derived artifact meets all specifications.

This approach is appealing because the user benefits from the method and the rationales are captured at a relatively low cost. The challenge is to provide a method that really does help the user without imposing excessive overhead or restrictions.

Apprentice. In this approach, the system generates rationales by essentially "looking over the designer's shoulder" and asking questions whenever it does not understand or disagrees with the designer's action. ADD,[15] for example, learns about the features that make a specific case different from a standard one. Whenever the designer's proposed action differs from ADD's expectations, it will ask the designer to justify or explain the difference. Later, it answers queries for design rationales by using both its domain knowledge and the designer-supplied justifications. Janus[16] also uses this approach.

Both the user and system benefit from this interaction. The user benefits if the system is right; the system learns something if it is wrong. In addition, rationales are captured in the context of use, when the relevant knowledge is fresh in the designer's mind and new

knowledge can be anchored in old. However, for this approach to be viable, it must create an initial knowledge base rich enough to understand most of the designer's actions and ask intelligent questions when it does not.

Automatic generation. In the simplest form of this approach, the system generates design rationales automatically from an execution history. An expert system that uses a trace of its rule invocations to explain the why and how of its actions is an example. This approach has the appeal of creating rationales at little cost to the user later and of being able to maintain consistent and up-to-date rationales. However, it also has the high initial cost of compiling the knowledge needed to construct the rationales. Furthermore, to go beyond the simple explanatory model based on execution traces, many issues at the core of machine-learning research must be resolved: what parts of the problem-solving trace must be captured and how, how to infer rationales from the trace, how to assess their relevance, and how to adapt them to the current situation.

How to access rationales

A design rationale system need not make rationales directly accessible to the user. However, accessibility lets the user directly examine, update, or revise rationales, making design activities more cost-effective and efficient. Such systems can be classified as *user-initiative* or *system-initiative*, depending on whether the user or the system initiates access.

In a user-initiative system, the user decides the parts of the design rationale to examine and when or how to look at them. These systems must help the user become aware of what exists and make it easy to find the desired parts. Most systems adopt simple versions of query and navigational aids (trees, zoomable graphical views), but many useful navigational tools such as the fish-eye view and guided tour are still not being used.

In a system-initiative system, on the other hand, the system decides when and how to present which parts of the rationales. Such systems must have enough knowledge to make intelligent decisions and must present rationales in ways that the user finds unobtrusive. Possible solutions to the second problem are the Janus critic[16] and ADD's apprentice[15] (described earlier). However, these solutions have only begun to address the issue of how to make a system understand the designer's actions enough to ask intelligent questions at appropriate times without seeming obtrusive. Much more research is needed before a system-initiative approach becomes viable.

Another critical issue associated with design rationale access is intelligent indexing. Designers will be disrupted if the system's response is slow, no matter how appropriate its timing might be. A special case of the tradeoff between representational formality and computational services (described earlier) is the tradeoff between the structure in rationale representation and ease of retrieval. Incremental formalization provides an interesting approach to the indexing problem. For example, MIT's mSibyl uses the formal representation of knowledge about mechanical artifacts to provide more intelligent indexing and matching of design rationales.

When the cost bearer is not the same as the beneficiary, providing a cost-effective system becomes more problematic. In fact, many groupware systems fail exactly because of this mismatch.

How to manage rationales cost-effectively

Like anything else, a design rationale system will not be used if the cost outweighs the benefits. The benefits are the services the system can provide. The cost is primarily the cost of producing the rationales in a form that can support the services. The fixed cost, such as building the system or the initial knowledge base, is not a big problem as long as the cumulative benefits from the system's use outweigh it. The bigger question is, who will bear the cost of producing design rationales for a particular artifact, and why?

The ideal cost bearer is the person who gets enough benefit to compensate for the cost. The interactive acquisition of the rationales in which the user and the system mutually benefit is an example, although even here the extra time and attention spent interpreting and answering questions is still a cost that requires compensation.

When the cost bearer is not the same as the beneficiary, providing a cost-effective system becomes more problematic. In fact, as Jonathan Grudin of the University of California, Irvine, points out, many groupware systems fail exactly because of this mismatch.[17] For example, most online meeting schedulers fail because they require people to maintain their local calendars online, even though the only beneficiaries are those responsible for scheduling the meeting. Grudin recommends either that the connection between the contributor and the benefit be made clear to all group members, or, better still, that there be a process along with the technology that delivers some benefit to the contributor. For example, whenever a designer benefits from part of a rationale, the system might allow the designer to send compliments to the contributor, which managers and others can see.

How to integrate the system

Integration is concerned with integrating the use of a design rationale system with all the varied aspects of design activities. It must be able to integrate

- Design rationales of multiple users despite heterogeneous mediums (video, audio, text) and/or platforms (Unix, Windows, Mac OS).
- Design rationales with other objects that serve different functions—for example, to link design rationales to objects in CAD or database modules.
- Different types of representation (object-oriented vs. rule-based, procedural vs. declarative, formal vs. informal).

Integration, like cost-effective rationale management, has not received much research attention in the context of design rationale systems. Only a few systems have attempted to address integration needs. More design rationale researchers should understand these types of integration and begin to adapt existing technologies to the systems they propose.

Among users. Because a design rationale system has many users, it should provide

some way to share objects or translate them over heterogeneous representations or systems. Exporting and importing through a file may work in some cases. Another possible solution is the wrapper approach,[18] in which each system builds a layer that handles any heterogeneous input, allowing participants to maintain their autonomy. Yet another possible solution is to communicate via a blackboard by publishing a portion of each participant's private scratchpad. Both options will require developing a common protocol and language or an extension of an API for exchanging design rationale information.

Among multimedia objects. Even when there is only one user, a design rationale system must be able to integrate various multimedia artifacts that must be accessed: design notebooks, sketchbooks, phone conversations, videos, CAD drawings, and e-mail. Phidias[16] addresses this problem by providing uniform access to artifacts through hypertext typed links, which the user can define. Phidias stores and manages these heterogeneous objects in a single hypermedia database, the Hyper-Object Substrate.

With design modules. A design rationale system must integrate well with other design components, such as CAD module databases and simulation packages. Because these components are often implemented on different platforms, integration also means achieving interoperability across heterogeneous platforms. Integration with other modules requires not only that these objects be linked or presented to the user but also that other modules be able to understand and manipulate them—at least enough for, say, the critic module to use design rationale objects and decide whether or not to interrupt the user.

Shared-DRIM[3] is one system that is well integrated with many other design modules, including Object Store, the Cosmos rule system, the Coplan constraint manager, the Dotstrean communication manager, the Gnomes geometric module, the Shared abstraction and integrity manager, and the Congen alternative generator. PTTT[19] provides an example of integration with standard operating procedures, a compound document editor, mail, a code management system, and a photo manager. When PTTT identifies a process problem, for example, it can display a standard operating procedure, which can then point to the stored information (experiments, process sheets) used to develop the process step. Examining the insights and problems in such an integrated system can aid the design of a future design rationale system.

AMONG THE SEVEN ISSUES I have identified, only a few have been explored in sufficient depth. Issues such as how to represent design rationales will not pose many new problems for developers of future design rationale systems. However, a few critical issues have been neglected. For example, I believe providing for cost-effective use, domain-knowledge generation, and integration give rise to many open research questions.

Designing a cost-effective system is one of the most urgent issues design rationale researchers face. Without a handle on this problem, the system, even with many sophisticated features, may not be used or, worse, may be counterproductive.[20] Neither do I believe that management is beyond the concern of research. The eventual goal of all research is to pave the way for practical systems. Of what practical value can engineering be if it is not cost-effective? The challenge is for researchers to build in an incentive structure that would obviate the need for a separate management structure. One approach is to provide a game-like interface for acquiring design rationales. Similar problems have been studied in other fields such as computer-aided instruction and machine learning. Future research in design rationale systems would benefit from these studies.

Another urgent need is for methods to produce formal design rationales at less cost. Incremental formalization is promising, but existing proposals are only a beginning. Keyword-based parsing can go only so far, and a robust technology for understanding natural language is not yet here. A restricted form of natural language with a menu-based interface might be a compromise worth exploring. Again, researchers would do well to examine the many relevant techniques and methods that have been explored in knowledge acquisition, reuse, natural language understanding, and machine learning.

Finally, integration is an important issue that can no longer be overlooked. Researchers in the management of design rationale systems can look at solutions that work in other areas, such as wrappers, mediators, OLE, and APIs.

By addressing these neglected areas, design rationale research can enhance its contribution to design research and begin to produce more esffective and economical systems.

References

1. J. Lee, "AAAI '92 Workshop on Design Rationale Capture and Use," *AI Magazine*, Vol. 14, No. 2, 1993, pp. 24–26.

2. J. Conklin and K. Burgess-Yakamovic, "A Process-Oriented Approach to Design Rationale," in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, eds., Lawrence Erlbaum Associates, Mahwah, N.J., 1995, pp. 393–428.

3. F. Pena-Mora, D. Sriram, and R. Logche, "Design Rationale for Computer-Supported Conflict Mitigation," *J. Computing in Civil Eng.*, Vol. 9, No. 1, 1995, pp. 57–72; URL for Shared-DRIM: http://ganesh.mit.edu/feniosky/shared_ pub.html.

4. S. Yamamoto and S. Isoda, "SOFTDA—A Reuse-Oriented Software Design System," *Proc. Compsac*, IEEE Computer Society Press, Los Alamitos, Calif., 1986, pp. 284–290.

5. J. Lee and K.-Y. Lai, "What's in Design Rationale?" *J. Human-Computer Interaction*, Vol. 6, No. 3, 1991, pp. 251–280.

6. M. Maher and A. Garza, "Case-Based Reasoning in Design," *IEEE Expert*, Vol. 12, No. 2, Mar./Apr. 1997, pp. 34–41.

7. R. Neches, W. Swartout, and J. Moore, "Enhanced Maintenance and Explanation of Expert Systems through Explicit Models of Their Development," *IEEE Trans. Software Eng.*, Nov. 1986, pp. 1337–1351.

8. F. Shipman and R. McCall, "Integrating Different Perspectives on Design Rationale: Supporting the Emergence of Design Rationale from Design Communication," Tech. Report 96-001, Center for the Study of Digital Libraries, Texas A&M Univ., College Station, Texas, 1996.

9. M. Klein, "Capturing Design Rationale in Concurrent Engineering Teams," *Computer*, Vol. 26, No. 9, Jan. 1993, pp. 39–47.

10. B. Chandrasekaran, A. Goel, and Y. Iwasaki, "Functional Representation as Design Rationale," *Computer*, Vol. 26, No. 9, Jan. 1993, pp. 48–56.

11. C. Potts and G. Bruns, "Recording the Reasons for Design Decisions," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, 1988, pp. 418–427.

12. J. Lee, "Incrementality in Rationale Management," *Proc. Requirements Eng. Symp.*, IEEE CS Press, 1992, p. 283.

13. G. Fischer and K. Nakakoji, "Making Design Objects Relevant to the Task at Hand," *Proc. AAAI '91*, AAAI Press/MIT Press, Cambridge, Mass., 1991, pp. 67–73.

14. I. Baxter, "Design Maintenance Systems," *Comm. ACM*, Vol. 35, No. 4, Apr. 1992, pp. 73–89.

15. A. Garicia and H. Howard, "Acquiring Design Knowledge through Design Decision Justification," *Artificial Intelligence in Eng. and Manufacturing*, Vol. 6, No. 1, pp. 91–109.

16. G. Fischer et al., "Making Argumentation Serve Design," in *Design Rationale Concepts, Techniques, and Use*, T. Moran and J. Carroll, eds., Lawrence Erlbaum Associates, 1995, pp. 267–294.

17. J. Grudin, "Groupware and Social Dynamics: Eight Challenges for Developers," *Comm. ACM*, Vol. 37, No. 1, Jan. 1994, pp. 92–105

18. M. Genesereth, "An Agent-Based Approach to Software Interoperation," Tech. Report Logic-91-6, Stanford Univ. Logic Group, Stanford, Calif., 1991.

19. D. Brown and R. Bansal, "Using Design History Systems for Technology Transfer," in *Computer Aided Cooperative Product Development*, D. Sriram, R. Logcher, and S. Fukuda, eds., Lecture Notes Series, No. 492, Springer-Verlag, New York, 1991, pp. 544–559.

20. S. Shum and N. Hammond, "Argumentation-Based Design Rationale: What Use at What Cost?" *J. Human-Computer Studies*, Vol. 40, 1994, pp. 603–652.

Acknowledgments

I thank all the workshop participants and organizers who contributed to the valuable discussions on which this article is partly based. I also thank the four anonymous *IEEE Expert* reviewers for their valuable and detailed comments. Finally, I thank Frank Halasz, who got me interested in the topic of design rationales while I worked with his group at the Xerox Palo Alto Research Center.

Jintae Lee is an assistant professor of decision sciences at the University of Hawaii, where he is leading the Process Interchange Format project to define a common language for process descriptions. He is also an active member of the MIT Process Handbook project to create an intelligent repository of process descriptions for reuse in process modeling and analysis. His research interests are understanding and improving knowledge sharing processes. Lee received a BA in mathematics from the University of Chicago, an MA in psychology from Harvard University, and a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology in 1991. He is a member of the IEEE, the AAAI, and the ACM. His address is Dept. of Decision Sciences, Univ. of Hawaii, 2404 Maile Way, Honolulu, HI 96825; jl@hawaii.edu.