# Identifying Quality-Requirement Conflicts

BARRY BOEHM and HOH IN, *University of Southern California*

*Without a well-defined set of quality-attribute requirements, software projects are vulnerable to failure. The authors have developed QARCC, a knowledge-based tool that helps users, developers, and customers analyze requirements and identify conflicts among them.*

Despite well-specified functional and interface requirements, many software projects have failed because they had a poor set of quality-attribute requirements. Finding the right balance of quality-attribute requirements is an important step in achieving successful software requirements and products. To do this, you must identify the conflicts among desired quality attributes and work out a balance of attribute satisfaction. The importance of this balance can be seen in examples that failed to find it:

- In the New Jersey Department of Motor Vehicles licensing system, engineers chose a fourth-generation language to satisfy software affordability and timeliness objectives, but the system failed because of performance-scalability problems.
- The initial development of the National Library of Medicine MEDLARS II system had a plethora of layers and recursions for portability and evolvability, but was eventually



0740-7459/96/$05.00 © 1996 IEEE

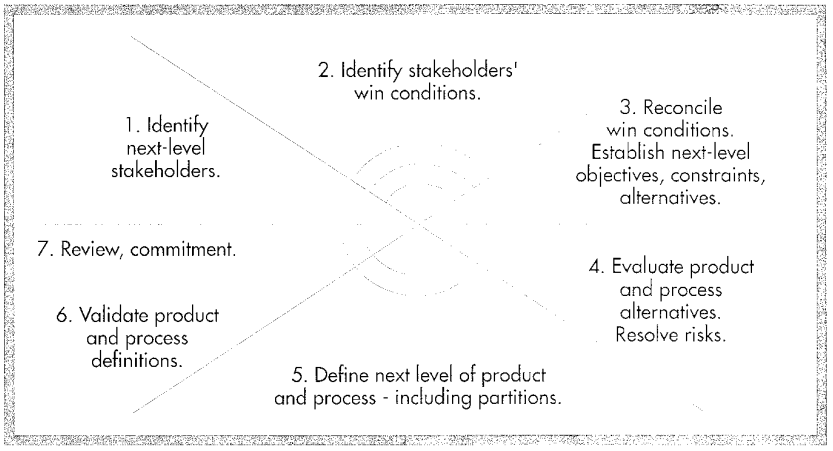



***Figure 1.*** *WinWin spiral model.*

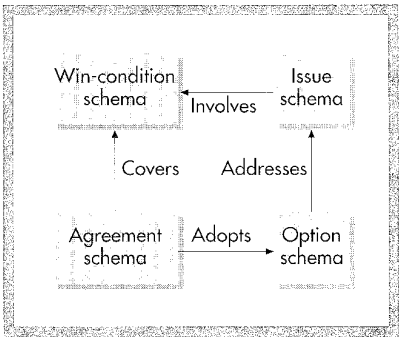



***Figure 2.*** *WinWin negotiation model.*

scrapped due to performance problems.

♦ The initial design of the ARPANet Interface Message Processor software — which was fortunately revised — focused on performance at the expense of evolvability through the design of an extremely tight inner loop.

Because it had an expert review team, the ARPANet problem was identified early and thus avoided. However, there is an overall scarcity of such software expertise. It would be valuable to capture the expertise that does exist and make it more broadly available through automated aids that analyze conflicts among software-quality attributes.

We have developed an initial version of such an aid. The Quality Attribute Risk and Conflict Consultant is a knowledge-based tool that can be used early in the life cycle to identify potential conflicts. QARCC operates in the context of the WinWin system,[1,2] a groupware support system developed at the USC Center for Software Engineering to determine software and system requirements as negotiated win conditions. QARCC works by examining quality-attribute tradeoffs involved in software architecture and process strategies. It may tell you, for example, that a layered architecture will improve portability, but usually at some cost in performance.

This article summarizes our experiences developing the QARCC-1 prototype using an early version of WinWin, and our integration of the resulting improvments into QARCC-2. In some cases, terminology has changed in the new version; these are noted where appropriate.

## THE WINWIN SYSTEM

To resolve quality-requirement conflicts, you must at least be able to

♦ identify and negotiate quality-attribute requirement conflicts and tradeoffs and

♦ diagnose quality attribute conflicts on the basis of early information.

We use the WinWin system to provide the first capability. For the second capability, the QARCC tool operates on the win conditions captured by the WinWin system to diagnose potential quality conflicts and tradeoffs among requirements early in the development process.

Figure 1 shows the WinWin spiral model, which serves as the basis for the WinWin system. The system uses Theory $W^3$ to generate the objectives, constraints, and alternatives needed by the spiral model. To meet its goal of "making everyone a winner," Theory W involves stakeholders in a process of identifying their quality-attribute win conditions (sector 2 in Figure 1) and reconciling conflicts among quality-attribute win conditions (sector 3).

Figure 2 shows the WinWin negotiation model's primary schemas and the relations among them. Stakeholders begin by entering their win conditions, using a schema provided by the WinWin system. If a conflict among stakeholders' win conditions is determined, an *issue schema* is composed, summarizing the conflict and the win conditions it involves.

For each issue, stakeholders prepare candidate *option schemas* addressing the issue. Stakeholders then evaluate the options, iterate some, agree to reject others, and ultimately converge on a mutually satisfactory option. The adoption of this option is formally proposed and ratified by an *agreement schema*, which includes a check to ensure that the stakeholders' iterated win conditions are indeed covered by the agreement.

In large systems involving several dozen or more win conditions, it is difficult to identify conflicts among them. To aid in manual and automated conflict assessment, the win-condition schema includes a slot for associating the win condition with elements of a taxonomy for the system domain.

WinWin also provides — and lets stakeholders tailor — domain taxonomies. Figure 3 shows an example for the software-engineering environment domain. The SEE domain taxonomy includes a "domain elements" section in the center and two relatively domain-independent parts: the infrastructure on the left and the attributes on the right. It is this attribute structure that the QARCC tool uses to identify potential quality-attribute conflicts among win conditions.

For each win condition that identifies a desired quality attribute, the QARCC tool uses a knowledge base to identify software architecture and

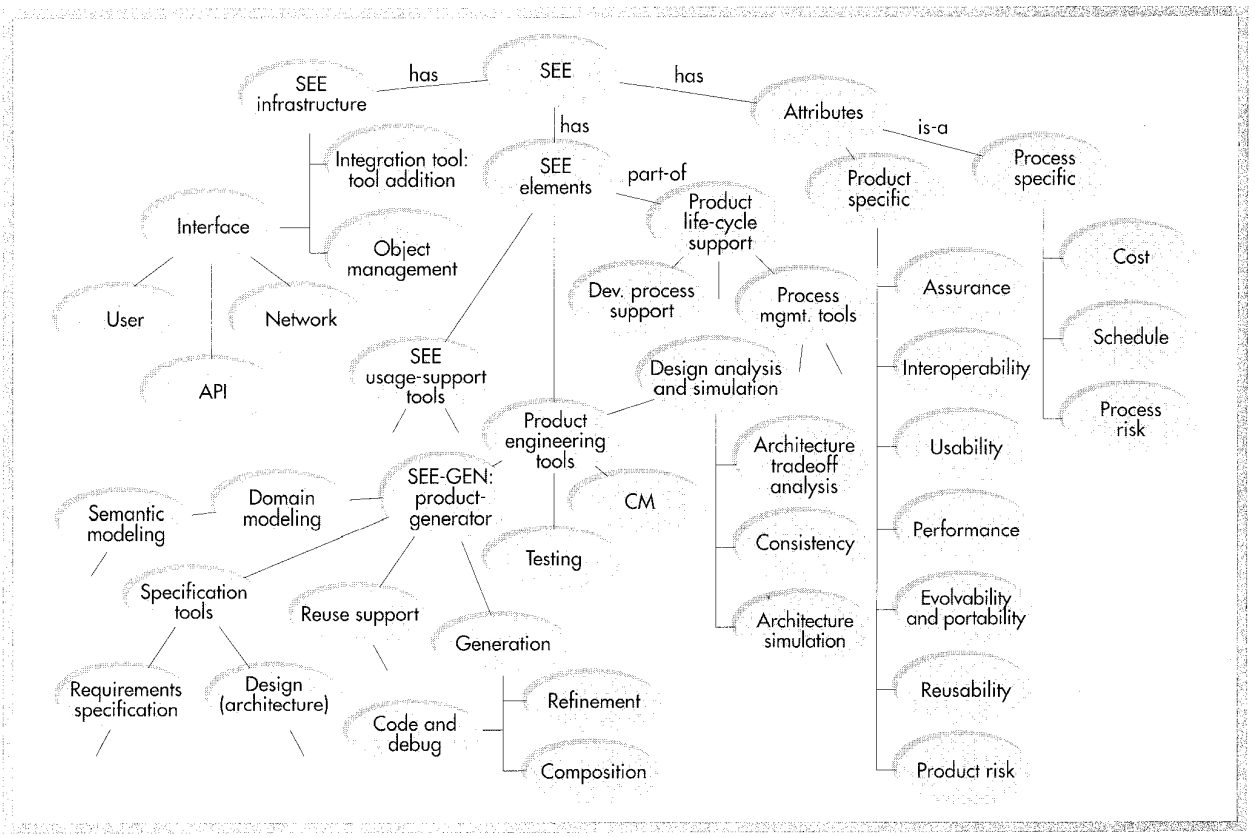



***Figure 3.*** *Example of WinWin domain taxonomy.*

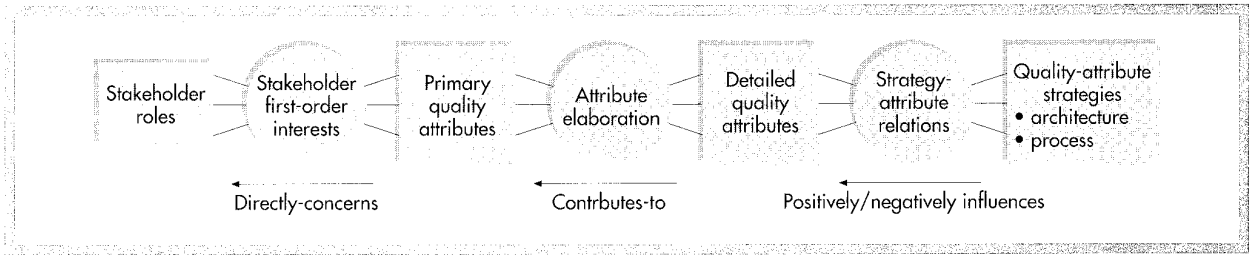



***Figure 4.*** *QARCC knowledge-base structure.*

process strategies for achieving the quality attribute. For each strategy, it uses another part of its knowledge base to identify potential conflicts with other quality attributes that could arise if the strategy were employed. It then provides suggestions about these potential quality-attribute conflicts and options for resolving them.

## QARCC KNOWLEDGE BASE

The context and information available for analyzing quality-attribute risks and conflicts early in the life cycle comes primarily from the prioritized requirements, as expressed by different system stakeholders' win-condition schemas. Overall, customers' primary concerns tend to focus on such attributes as cost and schedule, while users tend to be more directly concerned about such attributes as performance and assurance.

As the left side of Figure 4 shows, the structure of the stakeholders' first-order interests forms a part of the QARCC knowledge base, which includes a quality-attribute hierarchy similar to those in previous analyses.[4-6] The major difference here is that our hierarchy's highest level is connected to the quality attributes most directly valued by the various classes of stakeholders. For example, a maintainer tends to be primarily concerned with evolvability and portability and only secondarily concerned with development cost, schedule, and reusability, which tend to be primary concerns of customers and developers. This structure enables QARCC to associate quality-attribute risks and conflicts with the appropriate stakeholders. It thus flags potential concerns and provides stakeholders with advice for resolving them.

The other major component of the QARCC knowledge base, shown on the right of Figure 4, is a set of relationships between software architec-

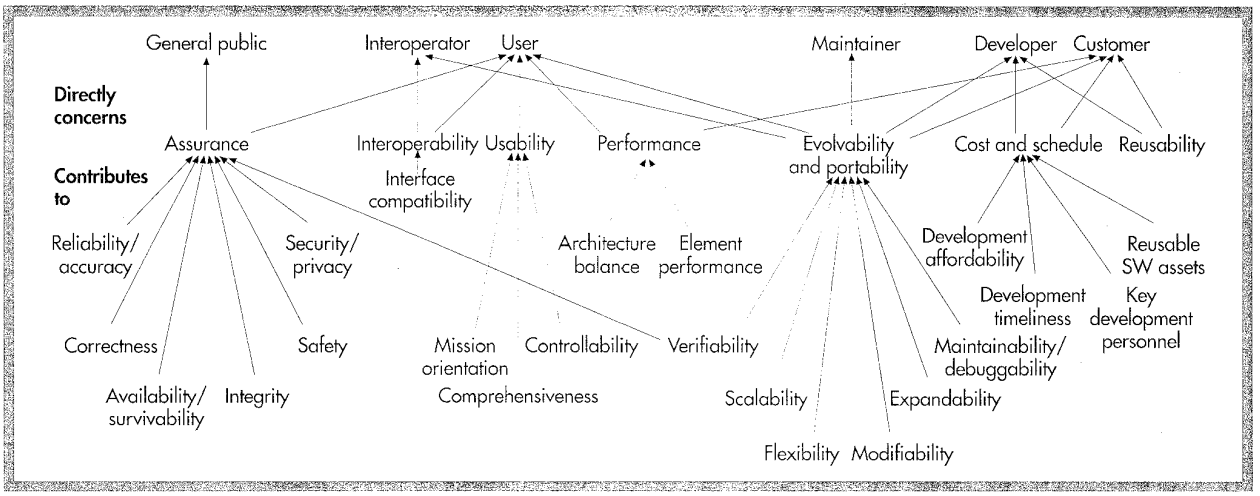



***Figure 5.*** *Mapping of stakeholders' primary concerns onto quality attributes.*

ture and process strategies and their usual effects on quality attributes. For example, a layered architecture has a positive influence on portability because a layer can hide platform dependencies; it has a negative influence on performance because in-line machine-dependent code is usually more efficient. Thus, using a layered architecture strategy to achieve portability will frequently cause a conflict with performance objectives.

## MAJOR COMPONENTS AND RELATIONS

The most frequent stakeholders are users, customers, developers, maintainers, interfacers, and the general public. Others could be product-line managers, testers, or subcontractors. Complex systems may have several different sets of users and customers.

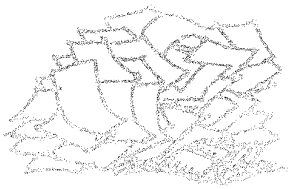
**First-order stakeholder interests.** We determined the primary-attribute win conditions for each stakeholder role through our experience in applying Theory W to complex, multistakeholder projects such as Army WWMCCS Information System, STARS, and the WinWin system itself. Figure 5 diagrams how we mapped the stakeholders' primary concerns to quality attributes. At the top of Figure 5 are the stakeholders and their primary requirements. Second-order stakeholder interests are also important (the developer cares about usability because

### IDENTIFYING QUALITY REQUIREMENTS: A COMPARISON

One initial approach to specifying quality attributes was the Requirements-Properties Matrix.[1] It provided a cross-impact matrix between the software functional requirements and the required properties or attributes, which served as a manual framework for identifying derived functional requirements implied by the attribute requirements. Several of the Rome Laboratory Quality Metrics reports — such as the one by James McCall and his colleagues[2] — provided checklists of attribute capabilities to be considered in requirements specifications, but did not address automated conflict analysis. A later Rome-sponsored study by Douglas Schaus developed a framework for an automated assistant for specifying quality software.[3]

Tom Gilb provides a framework for finding and specifying desired attribute levels in terms of solution specification, tagging, hierarchies, modularization, and cross-referencing, but no resolution aids are provided.[4] Steve Easterbrook provides a good conceptual framework for conflict resolution between domain descriptions with computer-supported negotiation.[5] He also provides an approach for resolving conflicts between different domain specifications, and provides an example using a library information-system specification. Lawrence Chung and his colleagues provide a good system framework for increasing traceability of quality attributes when changes in quality attributes or their importance, or design decisions and rationale, occur during the development process.[6] The system draws on domain knowledge to aid in assessing quality attributes, whereas our approach is domain-independent (although tailorable to specific domains).

Rick Kazman and his colleagues provide a five-step method for analyzing software architectures by analyzing three separate user-interface architectures with respect to the quality of modifiability.[7] They use representative operations to analyze the relationship between software architectures and quality attributes, but leave open the question of the sufficiency of the representative operations. Kazman and Len Bass explore the relationship between architectural "unit operations" and a method for deriving software architectures from eight quality-attribute requirements.[8] They provide a useful first-order conflict analysis of the interaction between the eight attributes, which we have used and extended in our analyses. However, their method of deriving architectures from requirements is somewhat oversimplified. Our assessment is that finding the right balance among conflicting quality attributes is too complex for simple algorithms, and that provid-



the user does), but these are generally addressed by negotiating first-order stakeholder win conditions.

**Attribute elaboration.** The lower set of arrows in Figure 5 show the next level of detail in the hierarchy. As the figure shows, the assurance attribute, which is a primary concern of users and the general public, may have several subattributes.

In many cases, it is sufficient to reason about attribute conflicts at the primary attribute level, but the lower levels are important at times — such as when conflicts arise between fault-tolerance data distribution for availability and restricted data access for security.

**Strategy-attribute relations.** Table 1 shows the general set of quality-attribute strategies in the knowledge base, organized into product and process strategies. We determined these strategies as a result of reviewing and filtering numerous studies of individual and multiple quality attributes.

Table 2 shows the elaboration of several architecture-based strategies for improving quality attributes, including top-level assessments of their effect on other quality attributes. For example, the input-checking strategy applies to several assurance subattributes, such as invalid data checking for reliability and unauthorized-access checking for security. Input checking also reinforces interoperability through validity and access checking across system interfaces. It reinforces usability by providing rapid feedback on invalid user inputs. On the other hand, the input-checking activities require additional code, memory, and execution cycles, and thus may conflict with the cost/schedule and performance attributes.

**Elaboration of attribute architecture strategies.** We are developing a formal structure for the quality-attribute architecture strategies summarized in Table 2. It is currently composed of a *definition* for each elementary strategy, *preconditions* to check whether or not the environments or situations are eligible for applying the strategies, *post-conditions* to describe the results or actions after applying the strategies, *effects* on quality attributes with rationale, and *options* for resolving quality-attribute conflicts.

Figure 6 shows the structure of elementary architecture strategies for quality attributes; Figure 7 shows three examples. The *preconditions* for the input-acceptability-check strategy are sets of candidate inputs and acceptability criteria. As the figure shows, the *effect* on assurance is positive, while the effects on performance, cost, schedule,

ing options and suggestions for stakeholders and architects is likely to have a higher payoff.

The object-oriented design patterns developed by Erich Gamma and his colleagues provide additional candidate-attribute strategies, particularly in the areas of evolvability/portability, interoperability, and reusability.[9] We are analyzing these to capture extensions to the QARCC knowledge base. Like our WinWin and QARCC system, William Robinson and Steve Fickas provide a model and a tool (called "Oz") that detects and resolves conflicts, and provides an interactive resolution-choice procedure and records of the negotiation process.[10] Their approach requires a domain-dependent knowledge base covering very detailed-level conflicts (such as conflicts of loan period in a library's system requirements). In contrast, our approach focuses on domain-independent conflicts involving high-level quality-attribute and architecture-strategy conflicts to achieve generality and scalability. A related and widely used approach for reconciling quality attributes is quality-function deployment.[11] It is a largely manual approach for which QARCC can provide complementary automated support.

## REFERENCES


1. B. Boehm, "Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design," *Proc. IFIP 74*, North Holland, Amsterdam, 1974, pp. 192-197.
2. J. McCall, P. Richards, and G. Walters, "Factors in Software Quality," Tech. Report 77CIS02, General Electric Command and Information Systems, Sunnyvale, Calif., 1977.
3. D. Schaus, "Assistant for Specifying Quality Software (ASQS) Mission Analysis," RADC-TR-90-348, Rome Laboratory, Rome, N.Y., Dec. 1990.
4. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, Reading, Mass., 1988.
5. S. Easterbrook, "Handling Conflict Between Domain Descriptions with Computer-Supported Negotiation," *Knowledge Acquisition*, Mar. 1991, pp. 255-289.
6. L. Chung, B. Nixon, and E. Yu, "Using Non-Functional Requirements to Systematically Support Change," *Proc. Second Int'l Conf. Requirements Eng.*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 132-139.
7. R. Kazman et al., "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. 16th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 81-90.
8. R. Kazman and L. Bass, "Toward Deriving Software Architectures From Quality Attributes," CMU/SEI-94-TR-10, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Penn., 1994.
9. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
10. W.N. Robinson and S. Fickas, "Automated Support for Requirements Negotiation," *AAAI-94 Workshop on Models of Conflict Management in Cooperative Problem Solving*, AAAI Press, Menlo Park, Calif., 1994.
11. L.P. Sullivan, "Quality Function Deployment," *Quality Progress*, June 1986, pp. 39-50.

**TABLE 1**
**GENERAL QUALITY-ATTRIBUTE STRATEGIES**

| Attribute | Product Strategies | Process Strategies |
|---|---|---|
| Assurance | Accuracy optimization, backup/recovery, diagnostics, error-reducing user input/output fault-tolerance functions, input checking, instrumentation, integrity functions, intrusion detection and handling, layering, modularity, monitoring and control, redundancy. | Failure modes and effects analysis, fault-tree analysis, formal specification and verification, inspections, penetration, regression test, requirements/design verification and validation, stress testing, test plans and tools. |
| Interoperability | Distributability, generality, integrity functions, interface specification, layering, modularity, self-containedness. | Interface change control, interface-definition tools, interface testing and analysis, interoperator involvement, specification verification. |
| Usability | Distributability (groupware), error-reducing user input/output, help/explanation, modularity, navigation, UI consistency, UI flexibility, undo, user-programmability, user-tailoring. | Prototyping, usage monitoring and analysis, user engineering, user-interface tools, user involvement. |
| Performance | Architecture balance, descoping, distributability, domain architecture-driven, faster hardware, instrumentation, optimization (code/algorithm), parallelism, pipelining, platform-feature exploitation. | Benchmarking, modeling, performance analysis, prototyping, simulation, tuning, user involvement. |
| Evolvability/ Portability | Distributability, generality, input assertion/type checking, layering, modularity, self-containedness, understandability, user-programmability, user-tailorability, verifiability, visibility functions. | Benchmarking, maintainer and user involvement, portability-evolution-vector specification, prototyping, requirement-evolution-vector specification and verification. |
| Cost/Schedule | Architecture balance, descoping, domain architecture-driven, modularity, reuse. | Design to cost/schedule, early error-elimination tools and techniques, personnel/management, process automation, reuse-oriented processes, user and customer involvement. |
| Reusability | Domain architecture-driven, portability functions. | Domain architecturing, reuser involvement, reuse-evolution-vector specification and verification. |
| All of Above | Descoping, domain architecture-driven, reuse (if strong with regard to attribute). | Analysis, continuous process improvement, incentivization, inspections, personnel/ management focus, planning focus, requirements/ design validation and verification, review emphases, tool focus, total quality management. |

and evolvability are negative.

We followed six steps for building the knowledge base for the strategy-attribute relations and quality-attribute strategies:

1. Identify primitive quality-attribute strategies. Table 1 summarizes the current working set of strategies.

2. For each identified strategy, analyze the *effects* on each of the other primary quality attributes as positive (+) or negative (-). For any pair of strategies with the same + and - pattern, combine them if they are sufficiently synonymous.

3. Define the *preconditions* and *post-conditions* involved in applying the quality-attribute strategies. These sharpen the strategy definitions, help validate the + and - assignments, and help identify more complex interactions.

4. Elaborate the more complex strategy-attribute relations (not just positive/ negative). For example, the monitoring and control strategy improves assurance at the cost of near-term performance, but also collects performance data supporting long-term performance improvement via tuning.

5. Formulate *options* to resolve the identified conflicts among quality attributes. Performance tuning is one example.

6. Update strategies based on experience.

## QARCC OVERVIEW

Figure 8 shows the QARCC concept of operation for identifying potential quality-attribute conflicts, flagging them for affected stakeholders, and suggesting options to resolve the conflicts.

QARCC is triggered by a stakehold-

**TABLE 2**
**QUALITY-ATTRIBUTE STRATEGIES AND RELATIONS: ARCHITECTURE STRATEGIES**

| Primary Attribute | Architecture Strategy | OtherAttribute Reinforcement | OtherAttribute Conflicts | Special Cases/ Comments |
|---|---|---|---|---|
| Assurance | Input checking | Interoperability, usability | Cost/schedule performance | |
| | Redundancy | | Cost/schedule, evolvability, performance, usability | |
| | Backup/recovery | | Cost/schedule, evolvability performance | |
| | Monitoring and control | | Cost/schedule, performance | Long-term performance enforcement via tuning |
| Inter-operability | Input checking | Assurance, usability | Cost/schedule, performance | |
| Evolvability /portability | Layering | Interoperability, reusability | Cost/schedule, performance | |
| | Modularity (platform-dependent functions) | Reusability, usability (displays) | Cost/schedule, performance | |

```
<definition>::=[<diagram>]Definition:<string>
<preconditions>::=[Preconditions:<string>]
<postconditions>::=[Postconditions:<string>]
<effects>::=[{-<quality-attributes>:{'('(+/-)[,<rationale-string>]')'}}]
<options>::=[Options:<string>
              [-Pros:<string>]
              [-Cons:<string>]]
<quality-attributes>::={<quality-attribute>[,<quality-attribute>]}
<quality-attribute>::=Assurance | Interoperability | Usability | Performance | Evolvability &
Portability | Cost & Schedule | Reusability
```

***Figure 6.*** *The structure of elementary architecture strategies for quality attributes.*

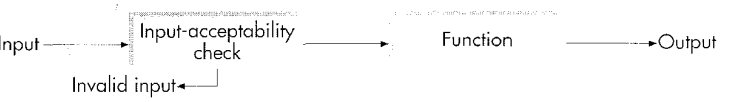
er entering a new win condition with a quality-attribute taxonomy element. Figure 9 shows screendumps from QARCC-1. For the attribute of portability in screen A, QARCC first considers its product and process strategies as given in Table 1 (such as layering to achieve portability). It then examines these strategies to search for potential conflicts with other attributes.

QARCC determines these potential conflicts from the portion of its knowledge base summarized in Table 2. For example, layering produces likely conflicts with cost/schedule and performance. (In QARCC-1, cost and schedule were combined under "development affordability" and performance was called "efficiency.") These are shown in the "potential conflict list" in screen B of Figure 9.

QARCC then uses the relationships shown in Figure 5 to identify the stakeholders affected by these potential conflicts (developer and customer for cost and schedule; user and customer for performance). For these stakeholders, QARCC pops up the "conflict advisor note" window (screen B) with the potential conflicts list generated by the new win condition. The list also enumerates any existing stakeholder win conditions that have conflicted attributes in their "taxonomy elements" slot. If no such win conditions exist, a "missing win condition" message is shown. For example, in screen B, development affordability has two existing win conditions — hohin-winc-5 and hohin-winc-9 — but assurance and usability have none.

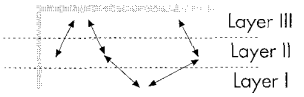
The stakeholder can select affected win conditions with the mouse and then click on the *create issue* button to have QARCC draft an issue schema shown in screen C. If no affected win conditions exist, the stakeholder can click on the *Create WinC* button to have QARCC draft a win-condition schema, shown in screen D.

An example of the draft material produced by QARCC is shown in the *Other's Comments* field of screen C, which cautions the stakeholders that affordability strategies such as reuse will



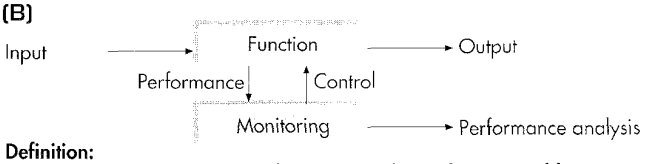

**Definition:**
An architectural composition that precedes a function by an acceptability check of its inputs.
**Preconditions:**
Candidate inputs, acceptability criteria.
**Postconditions:**
If valid, pass input into the Function, otherwise, indicate "Invalid Input" and exit.
**Effects on Quality attributes:**
- Assurance: (+) filters out unacceptable inputs.
- Performance: (-) input-check requires resources.
- Cost, Schedule: (-) more to specify, develop, and verify.
- Evolvability: (-) more to modify.

**(A)**




**Definition:**
A hierarchical architectural composition in which each layer can communicate only with the adjacent upwards or downwards layer.
**Preconditions:**
Interface and protocol between a layer and an adjacent layer, request to pass data and/or control from layer to layer.
**Postconditions:**
Data and/or control passed from layer to layer, or notification of interface/protocol violation.
**Effects on Quality attributes:**
- Evolvability, Interoperability, Portability, Reusability: (+) hides sources of variation inside interface layers.
- Performance: (-) requires more interfaces and data and/or control transfers via protocol.
- Cost, Schedule: (-) more to specify, develop, and verify; can reduce integration cost and schedule.

**(B)**




**Definition:**
An architectural composition that monitors the performance of function, controls the configuration or environment to stabilize the function (for example, to avoid buffer overflows), and/or reports the result for subsequent performance analysis.
**Preconditions:**
Monitoring instrumentation, control limits and algorithms.
**Postconditions:**
If the function is stable, checks the performance and reports it, otherwise stabilizes the function by controlling the configuration or environment.
**Effects on Quality attributes:**
- Assurance: (+) avoids undesirable states.
- Performance: (-) requires additional processing in short term; (+) improves performance in long term via system tuning.
- Cost, Schedule: (-) more to specify, develop, and verify.

**(C)**

***Figure 7.*** *Three examples of primitive quality-attribute architectural strategies: (A) input-acceptability check, (B) layering, and (C) monitoring.*

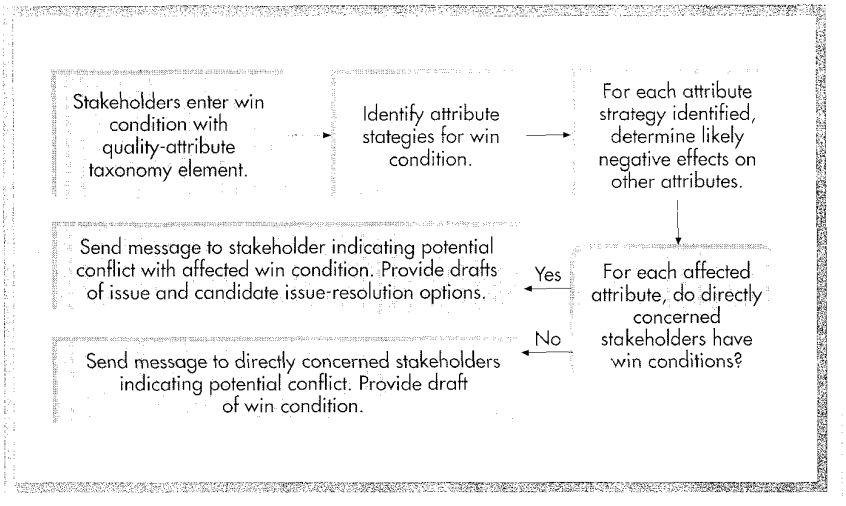



***Figure 8.*** *QARCC concept of operation.*

conflict with the portability win condition if the reused software is not portable.

By clicking the *Options* button at the bottom of screen C, the stakeholder can have QARCC draft a set of candidate resolution options. As the left window in screen E shows, the QARCC knowledge base generated six options to resolve the conflict between development affordability and portability:

- reduce or defer product functions;
- find, incorporate some relevant reusable software;
- find, engage expert performers;
- use design-to-cost process and identify lower priority features to defer if necessary;
- relax constraints on schedule,

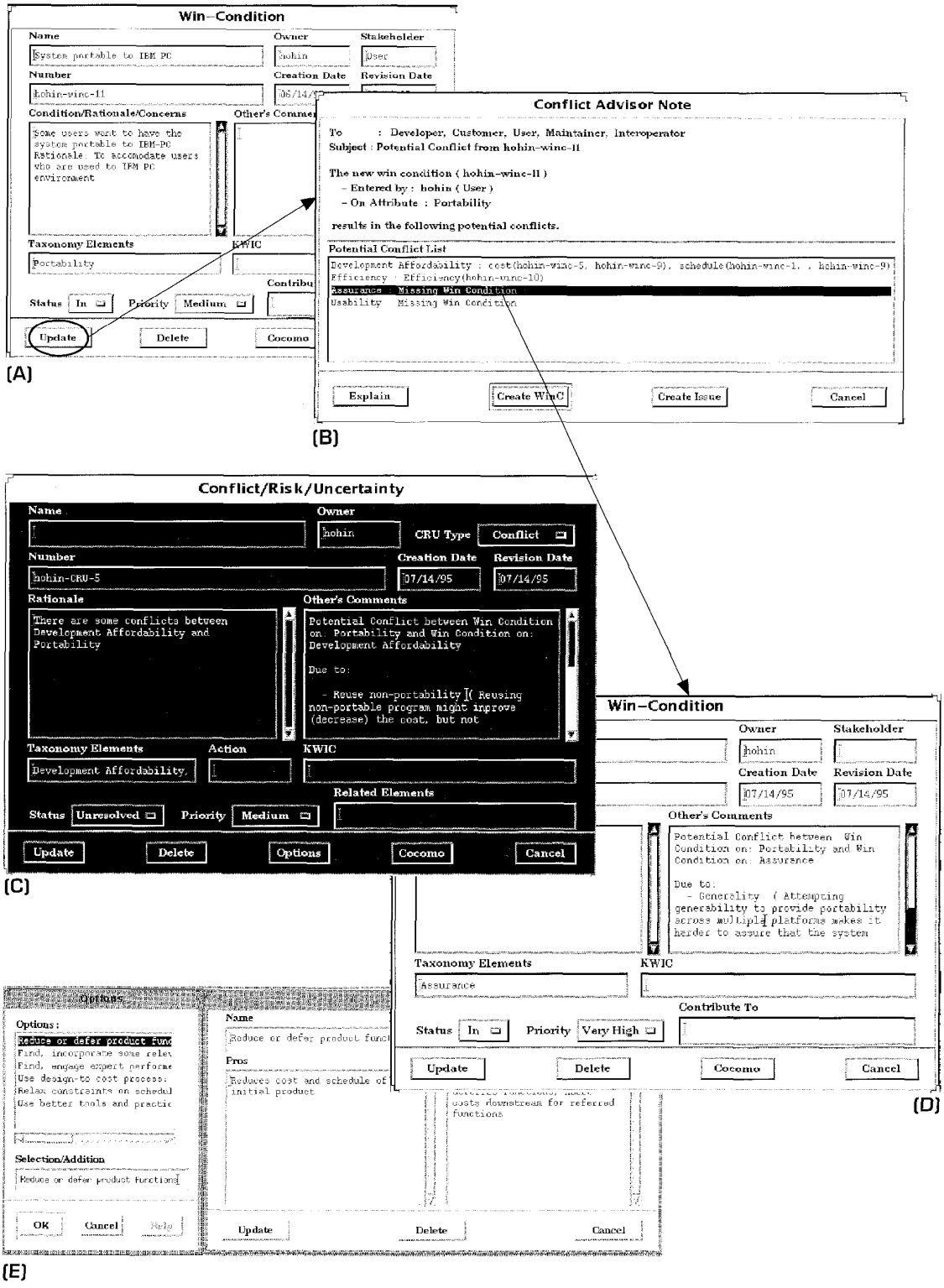



*Figure 9. An example of the initial implementation of QARCC.*

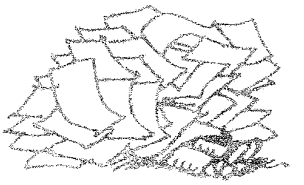


**TABLE 3**
**NUMBER OF CONFLICTS IDENTIFIED BY QARCC-1.**

| | Not found by QARCC | Found, significant | Found, insignificant |
|---|---|---|---|
| Conflicts found in WinWin user exercise | 0 | 2 | 0 |
| Conflicts not found in WinWin user exercise | 0 | 5 | 3 |

performance, hardware, and other attributes; or

- use better tools and practices.

As the options are generalized, stakeholders can tailor them to their special situations. QARCC also drafts pros and cons for the options (right window in screen E), helping the stakeholders evaluate the options and converge on a mutually satisfactory (win-win) option.

## EXPERIMENTAL RESULTS

QARCC-1 has been applied to several sample projects, primarily in satellite ground stations. In the experiment described here, we applied QARCC retroactively to the win conditions for a representative SEE to support a satellite-ground-station product line. The representative developer was a workstation vendor's CASE division, the representative user was a large aerospace ground-systems division, and the representative customer was the hypothetical US Space Systems and Operations Command.

The multistakeholder WinWin exercise generated 21 win conditions, including the following quality-related conditions:

- Initial operational capability (IOC) cost less than $7 million
- Full IOC delivery schedule within 25 months
- Interoperable SEE functions and tools
- Low development risk
- Low maintenance cost; easy to modify
- Commercializable middleware and commercially supported SEE to improve evolvability
- Broadly applicable across product line to improve evolvability

The main objective of the WinWin exercise was to determine the ability of WinWin to support renegotiation of a new win-win equilibrium solution when a new win condition was added to the base of 21 in-equilibrium win conditions. The new win condition, "support the development of multimission satellite ground stations," caused a cost and schedule conflict with the previously negotiated equilibrium. After determining that WinWin could successfully support such a renegotiation,[2] we decided to apply QARCC to the body of win conditions to see how many potential conflicts it would identify.

First, we wanted to see if QARCC would identify the two conflicts used in the renegotiation process. These were conflicts of cost/schedule with evolvability and interoperability: The stakeholders had rejected an option to recover cost and schedule by reusing legacy software that was deficient in evolvability and interoperability. Second, we wanted to see if QARCC would identify other potential conflicts, and if so, how many of them would have significant relevance to the satellite-ground-station system.

The results are shown in Table 3. QARCC found the two significant conflicts identified in the WinWin exercise. It also found eight more potential conflicts. Five of these were considered significant in the satellite-ground-station situation: conflicts of cost/schedule with assurance, performance, and reusability; and conflicts of interoperability and evolvability with performance. The conflicts not considered significant were those of evolvability with assurance and usability, and a conflict of cost/schedule with usability. We are reviewing these three "false alarm" situations to determine if the potential-conflict threshold for them was set too low for other situations as well. If so, we plan to drop them as being more time-consuming than beneficial.

From our initial experimentation, we concluded that QARCC can alert users, developers, customers, and other stakeholders to conflicts among their software-quality requirements and can help them identify additional, potentially important quality requirements. We also concluded that QARCC needs further refinement to avoid overloading users with insignificant quality-conflict suggestions. We are now refining the knowledge base to address more detailed quality attributes in a more selective fashion.

In our discussions with USC-CSE's industry and government affiliates who participated in demonstrations of QARCC-1, there was a strong consensus that it provided a useful framework for stakeholders to systematically resolve software quality-attribute conflicts. They also agreed that the semiautomated approach provided a good way to balance human skills and computer tools in addressing quality-tradeoff issues.

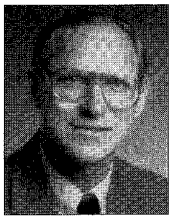
In our development and experimentation with QARCC-2, we are hoping to show that the QARCC approach is also scalable to large systems with many quality conflicts and that the effectiveness of the QARCC approach is largely domain-independent. ◆

## ACKNOWLEDGMENTS

This research is sponsored by the Advanced Research Projects Agency through Rome Laboratory under contract F30602-94-C-0195 and by the affiliates of the USC Center for Software Engineering: Aerospace Corporation, US Air Force Cost Analysis Agency, AT&T Bell Laboratories, Bellcore, Computer Science Corporation, Defense Information Systems Agency, E-Systems/Raytheon, Electronic Data Systems, Hughes Aircraft, Institute for Defense Analysis, Interactive Development Environments, Jet Propulsion Laboratory, Litton Data Systems, Lockheed/Martin, Loral Federal Systems, MCC Inc., Motorola, Northrop Grumman Corporation, Rational Software, Rockwell International, Science Applications International, Software Engineering Institute, Software Productivity Consortium, Sun Microsystems, Texas Instruments, TRW Inc., US Air Force Rome Laboratory, US Army Research Laboratory, and Xerox Corporation.

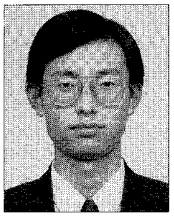
## REFERENCES

1. B. Boehm et al., "Software Requirements As Negotiated Win Conditions," *Proc. First Int'l Conf. Requirements Eng.*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 74-83.
2. B. Boehm et al., "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach," *Proc. 17th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 243-253.
3. B. Boehm. and R. Ross, "Theory W Software Project Management: Principles and Examples," *IEEE Trans. Software Eng.*, July 1989, pp. 902-916.
4. V. Basili and H. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proc. 9th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1987, pp. 345-357.
5. B. Boehm et al., *Characteristics of Software Quality*, TRW Series of Software Tech. Report TRW-SS-73-09, TRW Systems and Energy, Inc., Redondo Beach, Calif., 1973.
6. J. McCall, P. Richards, and G. Walters, "Factors in Software Quality," Tech. Report 77CIS02, General Electric Command & Information Systems, Sunnyvale, Calif., 1977.



**Barry Boehm** is the TRW Professor of Software Engineering and Director of the Center for Software Engineering at the University of Southern California. His current research involves the WinWin groupware system for software requirements negotiation, architecture-based models of software quality attributes, and the Cocomo 2.0 cost-estimation model.

Boehm received a BA in mathematics from Harvard University and an MS and PhD in mathematics from the University of California at Los Angeles. He is a fellow of the IEEE and the AIAA.



**Hoh In** is a PhD student at the Center for Software Engineering at USC. His research interests are in quality conflict resolution, including knowledge-based software requirements engineering, software architecture, design patterns, and software metrics and cost-estimation models.

Hoh received a BS and an MS in computer science from Korea University and won prizes for papers from the Korean Information Society and the Korean Academy Promotion Foundation.

Address questions about this article to Boehm or In at the Center for Software Engineering, USC, Los Angeles, Calif. 90089-0781; boehm@sunset.usc.edu or hohin@sunset.usc.edu. Additional information is available at http://sunset.usc.edu.