Identifying Requirements Conflicts and Cooperation:

How Quality Attributes and Automated Traceability Can Help

Alexander Egyed, *Teknowledge Corp.*

Paul Grünbacher, *Johannes Kepler Universität Linz*

Copyright © 2004 IEEE. Reprinted from *IEEE Software*. This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by sending a blank email message to pubs-permissions@ieee.org.

Requirements about software attributes have numerous complex and nontrivial interdependencies. Requirements conflict with each other when they make contradicting statements about common software attributes, and they cooperate when they mutually enforce such attributes. Because software developers rarely apply formal requirements specification techniques in practice, and because reliable techniques for natural language understanding aren't available, it's generally infeasible to automatically identify conflicts and cooperation among requirements. Furthermore, conflicts and cooperation exist only if the same part of the system realizes the requirements. Unfortunately, attributes often suggest otherwise, creating false conflicts.[1] Adding to the complexity is the fact that as systems evolve, new requirements emerge and existing ones change.

Here, we investigate how *requirements traceability* can help address these issues. RT is the ability to describe and follow the life of a requirement, forward and backward. We do this by defining and maintaining relationships to various artifacts created during system development, such as stakeholder needs, architectural or design elements, or source code.[2,3] RT facilitates communication, helps integrate changes, preserves design knowledge,[4] and supports quality assurance. It's thus beneficial throughout the entire life cycle but particularly important in iterative approaches, where stakeholders frequently introduce and change requirements.

Researchers have also recently used RT techniques to better understand and monitor persistent software attributes[1,5] such as reliability, scalability, efficiency, security, and usability. Analyzing requirements using software attributes can make conflicts and cooperation instances more obvious, because changes in

In software development and maintenance, identifying conflicts and cooperation among requirements is challenging. Fortunately, quality attributes can help. In addition, automated traceability techniques can eliminate falsely identified conflicts and cooperation efficiently.

 0740-7459/04/$20.00 © 2004 IEEE

WHY READ THIS ARTICLE?

To create software attributes that persist over a software system's lifetime, you must first understand how a system's attributes interact. Because every functional requirement is in effect a complex request for execution-time resources, such requirements can interact in unexpected ways that undermine complex and crosscutting properties such as performance, security, reliability, and other "-ilities." Identifying requirements conflicts that damage -ilities can be difficult even for a small, fixed set of requirements, and it can quickly become unmanageable if you add requirements without any strategy.

This article by Alexander Egyed and Paul Grünbacher addresses the requirements-phase PSA problem by defining a method to control growth in the complexity of requirements resolution, so that developers can recognize requirements conflicts that might damage -ilities before implementing them. Their method is based on the idea that requirements can conflict only when they translate into activities within the same region of the overall system. They show how requirements traceability can help you identify such overlaps without having to consider every possible (n^2) pairing of requirements. For methods such as agile programming that produce executable artifacts early in the development process, they discuss how you can partially automate such RT methods to further simplify the process of identifying potential conflicts. The net result is an ability to produce systems that are more easily and more provably able to preserve critical -ilities over their deployment life spans.

—Terry Bollinger, Jeffrey Voas, and Maarten Boasson, guest editors

quality often cause certain functional changes that in turn affect other quality attributes. For example, we know that checking user inputs increases correctness but also increases costs and decreases performance.

Our system builds on existing RT research (see the "Related Work" sidebar). It identifies requirements conflicts and cooperation using software attributes and eliminates false conflicts and cooperation automatically with the help of a trace analysis technique.

Video-on-demand's conflicting requirements

We demonstrate the need for our approach in the context of a simple video-on-demand system. The system we used provides capabilities for searching, selecting, and playing movies.[6] The "on-demand" feature supports playing a movie while concurrently downloading its data from a remote site. This system provides an interesting challenge because its complex computational logic is well hidden beneath a simple VCR-like user interface (with play, pause, and stop buttons). Both functional and nonfunctional issues are fundamental in defining the VOD system's requirements. Table 1 shows an excerpt of these requirements, including software attributes taken from ISO 9126.[7]

When analyzing trade-offs among these requirements, the developer must understand how the requirements affect one another. Fortunately, many rules of thumb exist to help with this process. One such rule is that a system's ability to recover from errors typically makes the system more useable by preventing abnormal program behavior—this would be an instance of cooperating requirements. A rule of thumb regarding conflicting requirements is that new functionality typically requires additional resources in computation and space, potentially conflicting with a requirement demanding high efficiency—for example, a short response time.

Figure 1 illustrates an example of two conflicting VOD system requirements in terms of their software attributes. R1 is a functionality requirement about playing a movie automatically after a user selects it (the user needn't press the play button). R6 is an efficiency requirement requesting a maximum delay of one second when starting a movie. Additional functionality usually lowers efficiency, and we can see a potential conflict between R1 and R6 when only looking at the conflicting attributes (see the red arrow in Figure 1a).

Software attributes define types of requirements, and existing work gives heuristics on conflict and cooperation among them.[8,9] Unfortunately, we can't decide automatically whether this potential conflict is true because there's no evidence that increased functional-

Related Work

Our work relates to existing research on investigating the complex interdependencies among system attributes.

Trade-off analysis

Barry W. Boehm and Ho Peter In have investigated the impact of quality changes in their Quality Attribute Risk and Conflict Consultant research for trade-off analyses. QARCC also relates different quality effects to various stakeholder groups. For example, performance and usability concern users whereas customers are interested in performance, maintainability, cost, and schedule. Developers would be mainly concerned with maintainability.[1]

Modeling persistent attributes

Lawrence Chung and his colleagues have discussed how properties such as modifiability and performance can be modeled as "soft goals" and how different architectural designs support these goals. In their approach, architectural decisions can be traced back to stakeholder goals.[2,3]

Aspect-oriented software development

AOSD is an approach supporting the separation of concerns. It provides explicit concepts to modularize crosscutting concerns and compose these with the system components.[4]

Automated techniques

Andrea Zisman and her colleagues have presented an approach for automatically generating and maintaining traceability relations based on rules. The artifacts and rules are described in XML and supported by a prototype tool.[5] The approach has also been applied to organizational models specified in *i** and software systems models represented in UML.[6]

References

1. B.W. Boehm and H. In, "Identifying Quality-Requirement Conflicts," *IEEE Software*, vol. 13, no. 2, 1996, pp. 25–35.
2. L. Chung, D. Gross, and E. Yu, "Architectural Design to Meet Stakeholder Requirements," *Software Architecture*, P. Donohue, ed., Kluwer, 1999, pp. 545–564.
3. L. Chung et al., *Non-Functional Requirements in Software Engineering*, Kluwer, 2000.
4. T. Elrad, R.E. Filman, and A. Bader, "Aspect-oriented Programming," *Comm. ACM*, vol. 44, no. 10, 2001, pp. 28–97.
5. G. Spanoudakis et al., "Rule-based Generation of Requirements Traceability Relations," *J. Systems and Software*, vol. 72, no. 2, 2004, pp. 105–127.
6. F.G. Cysneiros, A. Zisman, and G.A. Spanoudakis, "Traceability Approach for *i** and UML Models," *Software Eng. for Large-Scale Multi-Agent Systems Workshop Report* (SELMAS 03), to be published in *ACM Software Eng. Notes;* http://whitepapers.zdnet.co.uk/0,39025945,60093304p-39000629q,00.htm.

Table 1

Video-on-demand requirements

Requirement	Attribute
R0: Download movie data on demand while playing a movie	Functionality
R1: Play movie automatically after selecting from list	Functionality
R2: Display textual information about a selected movie	Functionality
R3: Pause a movie	Functionality
R4: Three seconds max to load movie list	Efficiency (including time behavior)
R5: Three seconds max to load textual information about a movie	Efficiency (including time behavior)
R6: One second max to start playing a movie	Efficiency (including time behavior)
R7: Use the major system functions (selecting, playing, pausing, stopping movie) without training	Understandability
R8: Allow users to stop a movie	Functionality
R9: (Re-)start a movie	Functionality
R10: Avoid image degradation caused by temporary network-load fluctuations	Reliability (including maturity)
R11: Only authorized users get access to movies	Security
R12: Automatically reestablish link to movie server within 5 seconds in case of failure during streaming	Recoverability

ity can't easily be satisfied within a given efficiency constraint (or the contrary). Because such evidence is hard to elicit automatically, it thus seems attractive to derive conflicts among requirements based on whether the requirements' software attributes contradict each other. If the attributes are indifferent toward one another or if they're cooperative, then a given set of requirements don't conflict.

Given that there are up to n^2 conflicts among n requirements, this approach significantly reduces the large number of potential conflicts. However, several disadvantages exist:

- Because attributes can affect each other in many obscure and nontrivial ways, they can identify potential conflicts and cooperation instances not applicable to a particular system (false positives).
- The number of potential conflicts, even if correct, could be enormous, leaving the engineer with the time-intensive and error-prone task of identifying true conflicts and cooperation.
- Because no single stakeholder has complete knowledge of all requirements, iden-

tifying true conflicts would involve numerous stakeholders as well.

An obvious simplification is to use domain-specific conflicts and cooperation among software attributes. This reduces—but doesn't eliminate—these disadvantages.

So how can we further eliminate false conflicts and cooperation? Figure 1b shows two requirements that don't conflict even though their software attributes do (a false conflict). R2 is a functionality requirement stating that users should be able to display information about selected movies. We investigate it against the previously discussed efficiency requirement of providing playback services within one second, and, on first glance, it seems there's the same kind of conflict between R2 and R6 as between R1 and R6 (Figure 1a). However, displaying movie information isn't done while selecting and starting a movie—it's implemented as an optional activity carried out either before or afterward—so R2 isn't restricted by the efficiency constraint.

If there's no trace dependency between two requirements, then they affect different parts of the system and thus can't affect one another, even if they're contradictory or cooperative. So, knowledge about trace dependencies helps eliminate false attribute conflicts and cooperation, but we still need to manually investigate the remaining ones. For example, there's still a potential conflict between R1 and R6 because a trace dependency exists—both relate to starting a movie.

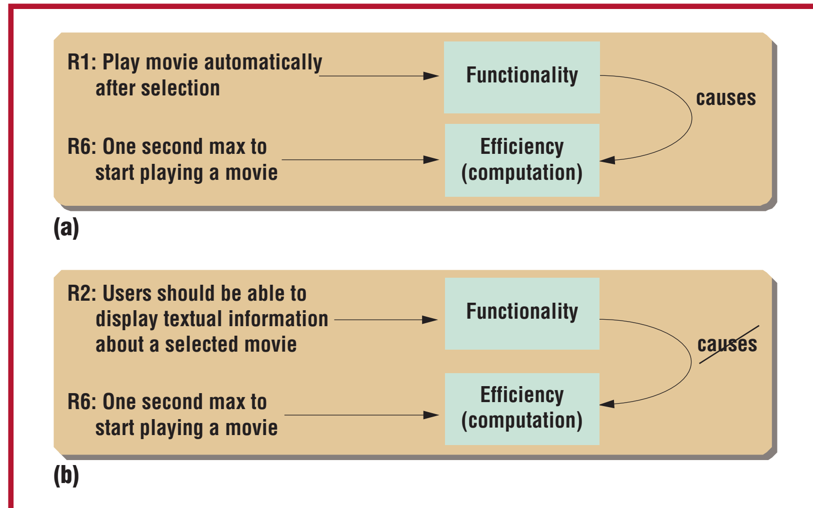

Figure 1. Conflicting requirements based on conflicting software attributes: (a) a potential conflict exists between adding functionality and maintaining efficiency; (b) the same potential conflict exists, but because these attributes don't apply to the same part of the system, this particular instance is a false conflict.

Investigating conflicts: Our approach

Our approach is suited for identifying requirements conflicts at any state in the life cycle as long as we have as input requirements, their attributes, and their traces. We assume that any two requirements conflict or cooperate only if their software attributes do the same and a trace dependency exists between them. If dependencies among requirements aren't available, then we generate them using a scenario-based approach to trace analysis that also requires test scenarios as input.[10]

Figure 2 depicts our approach schematically. It involves

1. Manually categorizing requirements into software attributes
2. Automatically identifying conflicts and cooperation among requirements based on their attributes
3. Automatically generating trace dependencies among the requirements (if not available)
4. Filtering out the attribute conflicts and cooperation instances between requirements where there's no trace dependency

The approach supports the incremental exploration of conflicts and cooperation by giving developers the freedom to change requirements and their test cases and refine the hierarchy of software attributes.

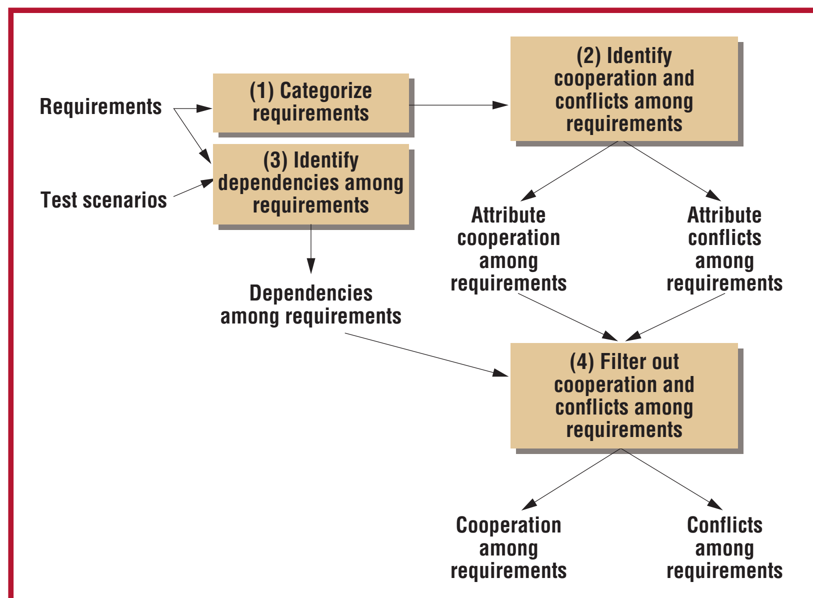

Figure 2. Approach for identifying conflicts and cooperation among requirements.

Table 2

Model of potential conflict and cooperation*

Requirement attribute	Effect							
	Functionality	Efficiency	Usability	Reliability	Security	Recoverability	Accuracy	Maintainability
Functionality	+	–	+	–	–	0	0	–
Efficiency	0	+/–	+	–	–	0	–	–
Usability	+	+/–	+	+	0	+	+	0
Reliability	0	0	+	+	0	0	0	0
Security	0	–	–	+	+	0	0	0
Recoverability	0	–	+	+	0	+	0	0
Accuracy	0	–	+	0	0	0	+	0
Maintainability	0	0	0	+	+	0	0	+

** + represents a positive effect; – represents a negative effect; 0 represents no effect*

Categorizing requirements

This first (manual) step is often part of a requirements definition process. We classify each identified requirement using a taxonomy of requirements (methods such as Volere[11] or international standards provide such taxonomies). Obviously, it helps to refine the taxonomies to address the specifics of particular domains. For example, we could refine efficiency into space and computational efficiency to support more specific reasoning.

Identifying conflict and cooperation among requirements

The next step is to analyze the requirements using a generic conflict and cooperation model such as the example shown in Table 2. This model takes into account that attributes might be indifferent to one another (0), cooperative (+), or conflicting (–). (We adapted Table 2 from a wide range of literature on nonfunctional requirements[12] and ISO 9126.[7])

Table 2 captures course-grained conflicts and cooperation. The rows represent requirement attributes; the columns represent their potential effects on other attributes and, consequently, other requirements. For example, a requirement that adds functionality likely has a negative effect on efficiency (–), a requirement that increases recoverability likely has a positive effect on usability (+), and a requirement that changes accuracy likely doesn't affect maintainability (0). A reverse effect negates the values in the table—for example, decreased security likely increases efficiency.

Where necessary, we can define subdimensions to better match the needs of specific domains or projects. For example, subdividing the efficiency dimension into space and computation efficiency means we can express conflicts and cooperation more precisely and can thus reduce false conflicts and cooperation. The values of categories should be the union of their subcategories, as in the case of efficiency shown in Table 3.

The model is useful when initially evaluating requirements conflicts and cooperation. However, it assumes worst- and best-case scenarios, so its values are conservative. In practice, this leads to identifying numerous false

Table 3

Refined cooperation and conflict model

Attribute	Effect								
	Functionality	Efficiency		Usability	Reliability	Security	Recoverability	Accuracy	Maintainability
		Space	Computation						
Space efficiency	0	+	–	0	–	0	0	–	–
Computation efficiency	0	–	+	+	–	–	0	–	–

conflicts and cooperation. Eliminating them by knowing about trace dependencies among requirements becomes critical.

Identifying dependencies among requirements

As systems evolve, it becomes increasingly ineffective to maintain traceability information. RT in practice often suffers from the enormous effort and complexity of creating and maintaining traces. It also suffers from incomplete trace information that can't assist engineers in real-world problems. We thus developed a tool-supported trace analysis technique[1,10] that's suited for evaluating the impact of new or changing requirements in large, highly complex systems that no single person can understand. Our trace analyzer uses testing to generate trace dependencies. If any two requirements affect the same part of a system, then their testing executes some of the same lines of code. Trace dependencies are created among requirements if their test scenarios execute the same or similar lines of code.

As input, the trace analyzer requires the user to associate requirements with test scenarios. Semantic and syntactic differences among requirements are irrelevant to determine trace dependencies. Only the difference between any given requirement and the system is relevant (n differences instead of n^2). Furthermore, using informal language isn't a problem because a requirement's meaning is captured in the form of attributes and test scenarios only.

The trace analyzer's key benefit is that it separates concerns during trace analysis so that you don't have to understand the relationships among all requirements. Rather, you need only understand the individual relationship between one requirement and its attributes and test scenarios—one requirement at a time, independently. The downside is that the trace analyzer detects trace dependencies among requirements only if it can map them to the system. Consequently, this approach applies only to product requirements—it excludes process-related requirements such as budgets or schedules.

Consider again the three VOD requirements discussed in Figure 1. Having identified test scenarios for these requirements, it's then straightforward to execute the scenarios. Figure 3 depicts this schematically in the form of arrows representing execution paths. For example, we can test the efficiency requirement demanding that a movie start within one second by clicking on the VOD player's "start movie" button and monitoring its execution path (the upper-left blue path in Figure 3). The other two requirements follow their own execution paths during testing.

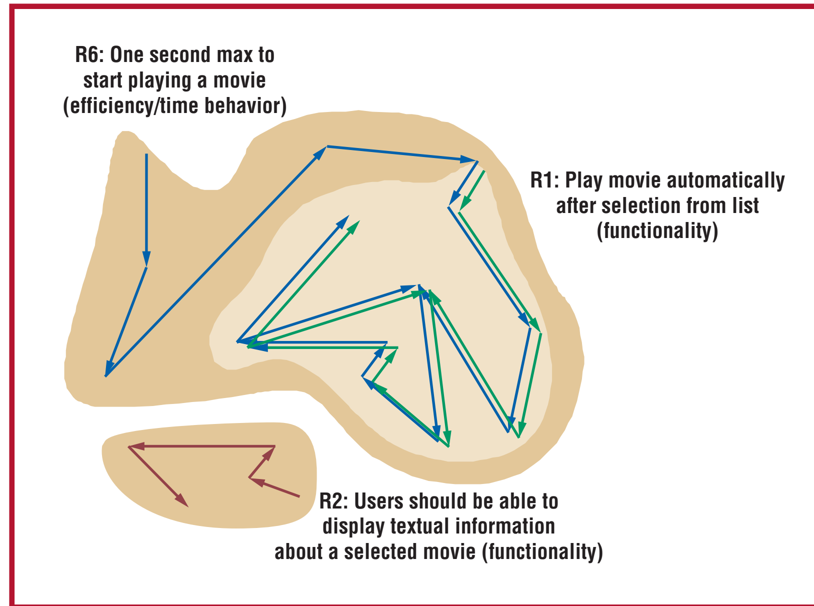

Figure 3. Execution paths (footprints) of three VOD requirements.

If more than one test scenario exists for a requirement, then its execution path is simply the combination of all individual paths (where the ordering is irrelevant). Once testing is complete, we infer trace dependencies among the three requirements in Figure 3 through their overlapping execution paths (called *footprints*).[1,10] Figure 3 shows that R1 and R6's footprints overlap. This implies some trace dependency between the efficiency (R6) and functionality (R1) requirements. There's no overlap between the footprints for R6 and R2, implying there's no trace dependency.

Filtering cooperation instances and conflicts among requirements

We now use knowledge of trace dependencies to disregard attribute conflicts and cooperation that affect different parts of the system. We interpret the trace dependency's meaning based on how much the requirements' footprints overlap. The overlap doesn't indicate conflict or cooperation, but knowing that there's no overlap lets us eliminate falsely identified conflicts or cooperation.

Figure 4 depicts different possible overlaps. The two extremes are a complete overlap in

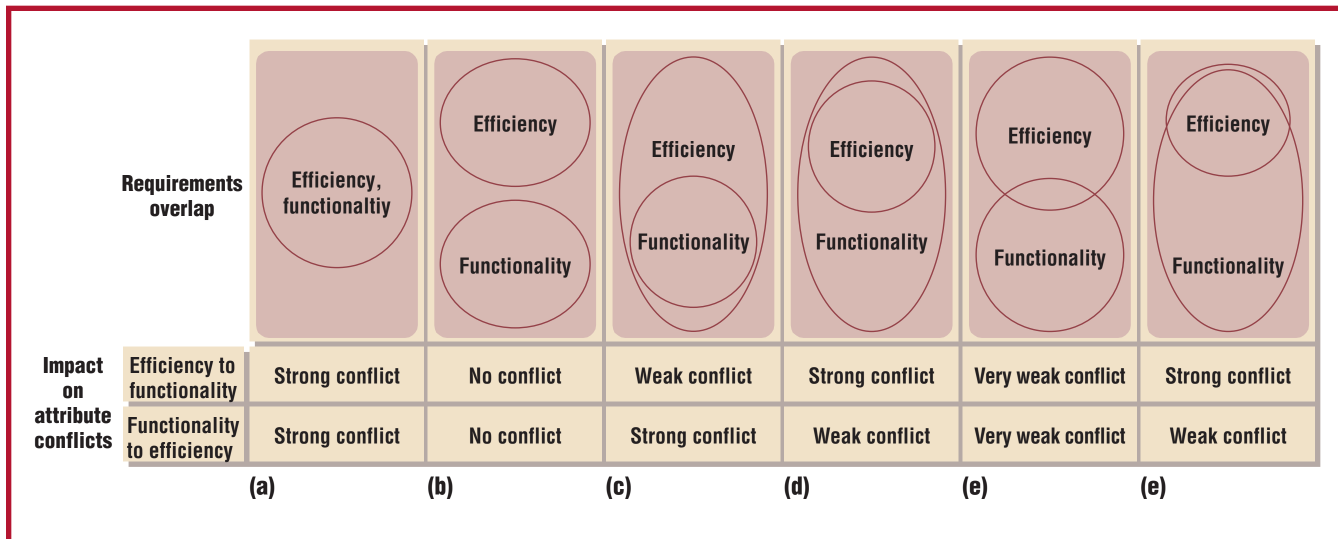

Figure 4. Overlaps among requirements and the impact on attribute conflicts: (a) complete overlap; (b) no overlap; (c) subset overlap, where functionality is a subset of efficiency; (d) subset overlap, where efficiency is a subset of functionality; (e) a small amount of overlap; (f) strong overlap.

the footprints of two requirements (Figure 4a) or no overlap (Figure 4b)—we found that the latter occurred frequently but the former did not. A complete overlap among requirements implies that the requirements are either identical or so closely intertwined that their execution can't distinguish them. Given that requirements define their scope and context rather arbitrarily, it's more likely that requirements overlap only partially. In this case, we distinguish between two scenarios: a footprint being a subset of the other (Figure 4c and 4d) and both footprints intersecting with one another but also having parts not shared (Figure 4e and 4f). Both scenarios are very likely, but the former has the advantage that the overlap is complete in one direction.

For example, R6 and R1 in Figure 3 overlap such that R1's footprint is a subset of R6's footprint. Thus, the functionality "play movie automatically after selection" (R1) overlaps fully with the efficiency requirement (R6). This implies that it must execute in less than one second to satisfy the efficiency requirement. If the functionality requirement's footprint doesn't overlap completely, then this would weaken the trace dependency in that only a part of the functionality must execute in less than one second but we don't know which part of the functionality that is. As another example, R6 doesn't overlap with R2 ("display textual information about movies"), implying that the efficiency requirement doesn't constrain functionality.

The trade-off among requirements is most meaningful if footprints overlap fully or not at all. Partial overlaps represent gray zones for the trade-off analysis. For example, R1's footprint is a subset of R6, implying that R1 overlaps with only part of R6. Although R6 describes an efficiency requirement, we don't know which subset of that requirement this particular part describes. So, the less overlap between any two requirements, the less meaningful the trace dependency between them.

We measure the weight of trace dependencies by the percentage of their overlap. For example, R1 overlaps fully with R6, implying a trace dependency with a weight of 100. On the contrary, R1 doesn't overlap with R2, so its trace dependency has a weight of 0. In between, R6 overlaps roughly 60 percent with R1, which implies that 60 percent of R6's code overlaps with R1.

Figure 4 also displays the effect of different overlaps (that is, weighted dependencies) in case of efficiency and functionality attribute conflicts. If the efficiency and functionality requirements overlap fully, the attribute conflict applies. However, if the requirements don't overlap, their attribute conflicts can't be requirements conflicts. Depending on directionality and the degree of overlap, the requirements conflict weakens.

Trade-off among multiple requirements

Requirements express expectations they *want* to satisfy. For example, the efficiency requirement R6 wants a one-second response time when starting a movie. The functionality requirement R1 wants a feature for playing a movie after selecting it (see the right side of

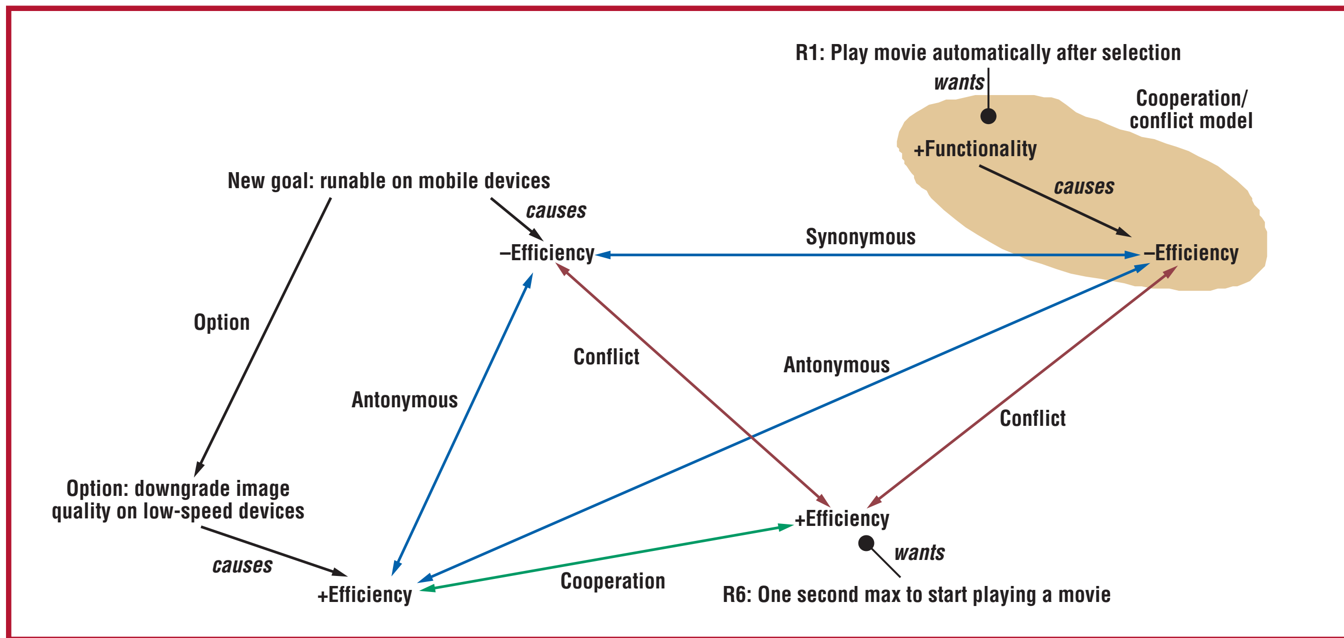

Figure 5. Trade-off among multiple requirements.

Figure 5). Requirements conflict and cooperate, however, on the basis of causal relationships among them—as such, there's no conflict in that R1 wants more functionality, which likely reduces efficiency (that is, this is just an effect). However, there's a potential conflict in that R1 *causes* less efficiency while R6 *wants* more efficiency.

Figure 5 explores this conflict in the context of potential new requirements. For instance, a stakeholder might want to also have the VOD run on handheld, mobile devices. This added flexibility likely causes less efficiency because handheld devices are computationally less powerful. Thus, this new requirement potentially conflicts with R6 because it will become harder to satisfy the one-second response time (note that the new goal affects the entire VOD system, so there's a trace dependency between the new requirement and both R1 and R6).

Our automated trace-analyzer technique doesn't require us to write code for this new requirement to eliminate false positives. The developer only hypothesizes which code the new requirement affects. The trace analysis then informs the developer about the new requirement's impact on all other requirements.

In the example, the relationship between the new requirement and the functionality requirement R1 is particularly interesting. Both cause less efficiency, which seems negative, but it'd be incorrect to define this as a conflict because less efficiency isn't necessarily a problem (neither wants more efficiency). Therefore, we define both requirements as having synonymous effects on the same part of the system (that is, they similarly lessen efficiency).

Now consider how to resolve the efficiency conflict between the new requirement and R6. One option is to reduce image quality on low-speed devices, which causes more efficiency. Clearly, this option cooperates with R6 in satisfying the one-second response time because the option causes more efficiency. The effect of the option, however, contradicts the functionality requirement and the new goal. As such, we can say they have antonymous effects on the same part of the system.

Conflicts, cooperation, and synonymous and antonymous effects are computed automatically. The example, however, demonstrates that it might be easier to define only a requirement's effect and not its attributes. For example, it's somewhat difficult to categorize the new goal as an attribute. We thus only define its effect of reducing efficiency and ignore its attribute. This gives the user added flexibility in how to define attributes and their effects.

Our approach is conservative in that it eliminates only false conflicts and cooperation. It's also highly scalable—it doesn't require understanding the interdependencies among requirements, because the input of attributes and test scenarios for requirements

can be defined independently for every requirement. Consequently, our approach generates a reduced, weighted list of potential conflicts and cooperation that is significantly shorter than the initial list. Although up to n^2 potential conflicts might exist among requirements, experience reports have shown few factual ones. Our largely automated and tool-supported approach thus spares users the costly and highly error-prone exploration of many false conflicts and cooperation. Future work will extend the list of existing software attributes to include subcategories, as discussed in our example of subcategorizing efficiency into space and performance. Furthermore, we intend to investigate how to complement our requirements traceability approach with other approaches, since our approach requires a testable system that isn't readily available early on.

About the Authors

Alexander Egyed is a research scientist at Teknowledge Corp. His research interests include requirements engineering, incremental and iterative software modelling (transformation and analysis), traceability, and simulation. He received his PhD in computer science from the University of Southern California. He is a member of the IEEE, IEEE Computer Society, ACM, and ACM SIGSOFT. Contact him at Teknowledge Corp., 4640 Admiralty Way, Ste. 1010, Marina Del Rey, CA 90292; aegyed@teknowledge.com.

Paul Grünbacher is an associate professor at Johannes Kepler University and a research associate at the Center for Software Engineering at the University of Southern California. His research interests include requirements engineering, computer-supported cooperative work, and automated software engineering. He received his PhD in computer science and economics from the University of Linz. He is a member of the IEEE, IEEE Computer Society, ACM, and ACM SIGSOFT. Contact him at Systems Engineering and Automation, Johannes Kepler Universität Linz, Altenbergerstr. 69, 4040 Linz; paul.gruenbacher@jku.at.

IEEE Pervasive Computing...

delivers the latest developments in pervasive, mobile, and ubiquitous computing. With content that's accessible and useful today, the quarterly publication acts as a catalyst for realizing the vision of pervasive (or ubiquitous) computing Mark Weiser described more than a decade ago—the creation of environments saturated with computing and wireless communication yet gracefully integrated with human users.

Editor in Chief: M. Satyanarayanan
Carnegie Mellon University

Associate EICs: Roy Want, Intel Research; Tim Kindberg, HP Labs; Gregory Abowd, Georgia Tech; Nigel Davies, Lancaster University and Arizona University

UPCOMING ISSUES:

- ✔ Energy Harvesting and Conservation
- ✔ The Smart Phone
- ✔ Ubiquitous Computing in Sports
- ✔ Rapid Prototyping

SUBSCRIBE NOW! www.computer.org/pervasive/subscribe.htm

References

1. A. Egyed and P. Grünbacher, "Automating Requirements Traceability: Beyond the Record & Replay Paradigm," *Proc. 17th IEEE Int'l Conf. Automated Software Eng.* (ASE 02), IEEE CS Press, 2002, pp. 163–171.
2. O.C.Z. Gotel and A.C.W Finkelstein, "An Analysis of the Requirements Traceability Problem," *Proc. 1st Int'l Conf. Requirements Eng.* (ICRE), IEEE CS Press, 1994, pp. 94–101.
3. B. Ramesh and M. Jarke, "Toward Reference Models for Requirements Traceability," *IEEE Trans. Software Eng.*, vol. 27, no. 1, 2001, pp. 58–93.
4. N. Medvidovic et al., "Bridging Models across the Software Lifecycle," *J. Systems and Software*, vol. 48, no. 3, 2003, pp. 199–215.
5. A. Egyed and P. Grünbacher, "Towards Understanding Implications of Trace Dependencies among Quality Requirements," *Proc. 2nd Int'l Workshop Traceability in Emerging Forms of Software Eng.* (TEFSE 2003), 2003; www.soi.city.ac.uk/~gespan/paper2.pdf.
6. K. Dohyung, "Java MPEG Player," 1999, http://peace.snu.ac.kr/dhkim/java/MPEG.
7. *ISO/IEC-9126, Software Product Evaluation—Quality Characteristics and Guidelines for Their Use*, ISO, 1991.
8. B.W. Boehm and H. In, "Identifying Quality-Requirement Conflicts," *IEEE Software*, vol. 13, no. 2, 1996, pp. 25–35.
9. L. Chung et al., *Non-Functional Requirements in Software Engineering*, Kluwer, 2000.
10. A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis," *IEEE Trans. Software Eng.*, vol. 29, no. 2, 2003, pp. 116–132.
11. S. Robertson and J. Robertson, *Mastering the Requirements Process*, Addison-Wesley, 1999.
12. L. Chung, D. Gross, and E. Yu, "Architectural Design to Meet Stakeholder Requirements," *Software Architecture*, P. Donohue, ed., Kluwer, 1999, pp. 545–564.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.