Proceedings of the 2nd International Workshop on Traceability In Emerging Forms of Software Engineering (TEFSE), co-located with ASE 2003, Montreal, Canada, October 2003.

Towards Understanding Implications of Trace Dependencies among Quality Requirements

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA
aegyed@acm.org

Paul Grünbacher
Systems Engineering and Automation
Johannes Kepler University
4040 Linz, Austria
gruenbacher@acm.org

Abstract

Understanding the implications of trace dependencies among quality requirements is necessary in critical engineering activities such as architectural risk assessment. In this paper we will first briefly summarize our scenario-based approach to generating trace dependencies and then demonstrate how to "add" meaning to the created trace dependencies in order to understand their implications. The paper also discusses automated support for trade-off analysis, and a brief discussion of related work.

1. Introduction

Many critical risks in software engineering are architectural [1] and deal with system properties like performance, reliability, or security [11]. Architectural risks have to be considered in particular when new requirements have to be assessed or changes to existing requirements are made. Assessing architectural risks however is challenging and relies on understanding the complex relationship between requirements, desired system properties, and architectures during development and maintenance [10, 12].

Requirements Traceability (RT) [9, 13] aims at understanding the complex relationship between different development artifacts. However, the approach suffers from the enormous effort and complexity of creating and maintaining traceability information [14]. Although numerous industrial-strength tools are available to manage trace links, there is still a strong need to automatically create trace links.

In our previous research we have thus been developing techniques and tools for automating the creation of trace dependencies between various models in the software life-cycle [6, 7]. Using our scenario-based approach in the context of the discussed problem to relate requirements, architectures, and quality properties assumes that we have

(a) requirements of a system expressed in an arbitrary notation;
(b) the architecture and implementation of a system
(c) quality attributes expressed through the requirements and architecture (either elicited from the requirements or reverse engineered from the architecture through some other means); and
(d) scenarios how to validate these requirements/ architecture/quality artifacts.

As a result of our approach we get various trace dependencies among artifacts, such as traces among functional artifacts (requirements, architecture), traces between functional and quality (non-functional) requirements, as well as traces among quality requirements.

Although the automatic creation of these dependencies is a big advancement over traditional traceability techniques we are still facing the challenge on how to interpret and use the created information in real-world engineering scenarios. We have to understand the meaning and implications of these trace dependencies and also need to find out how related artifacts affect one another to identify potential conflicts and risks. For example, requirements may be orthogonal, or they may positively or negatively reinforce each other.

The manual investigation of all trace dependencies however is tedious and error prone. In a real-world system there are likely thousands of links one would have to investigate in order to find potential conflicts. We are thus proposing a technique that (a) defines the implication of trace dependencies based on the meaning of the requirements they bridge; and (b) defines the positive and negative impact requirements changes may have.

2. Scenario-Based Trace Dependencies

2.1 Overview

Trace dependencies describe relationships between different artifacts such as requirements, designs, assumptions, rationale, system components, source code etc. [13]. They help engineers in answering real-world questions such as "Why is this requirements here?", or "What happens if I change this design element?". Traces

are not static but living entities in an iterative modeling process. For example, if a requirement led to the implementation of some source code then there should exist a trace dependency. If the requirement changes then the source code is most likely affected and vice versa.

The Trace Analyzer [7] automatically defines trace dependencies through shared use of source code. This means that if two requirements depend on subsets of the system's source code then a trace dependency exists if and only if those subsets overlap. The technique relies on known or hypothesized dependencies between *sets of* model elements and code where there is still uncertainty about the correctness of these dependencies. It helps in identifying relationships between *individual* model elements and code with a higher certainty about correctness.

2.2 Video-On-Demand System

The Trace Analyzer technique has been applied previously to a Video-On-Demand System [5] as described in [6]. This system allows searching for movies, selecting, and playing movies concurrently while downloading its data from a remote site. A shortened list of functional and quality requirements is published in [6]. Important requirements of the system include (requirements categories in brackets are taken from [11]):

r0: Download movie data on-demand while playing a movie *(Functionality)*

r1: Play movie automatically after selection from list *(Functionality)*

r2: Users should be able to display textual information about a selected movie *(Functionality)*

r3: User should be able to pause a movie *(Functionality)*

r4: Three seconds max to load movie list *(Efficiency/Time behavior)*

r5: Three seconds max to load textual information about a movie *(Efficiency/Time behavior)*

r6: One second max to start playing a movie *(Efficiency/Time behavior)*

r7: Novices should be able to use the major system functions (selecting movie, playing/pausing/stopping movie) without training *(Understandability)*

r8: User should be able to stop a movie *(Functionality)*

r9: User should be able to (re) start a movie *(Functionality)*

r10: Avoid image degradation caused by temporary network-load fluctuations *(Reliability/Maturity)*

r11: Only authorized users should get access to movies *(Security)*

r12: System should automatically re-establish a link to the movie server within 5 seconds in case of failure during streaming *(Recoverability)*

Our approach requires the existence of usage scenarios that can be tested against the code to identify trace dependencies. Commercial tools for monitoring the execution of a software system are readily available which report the lines of code, methods, and classes executed while testing a scenario. Table 1 below lists test scenarios[1] defined for this case study and shows what requirements the test scenarios apply to. The right part of the table shows the Java classes that get executed while testing various usage scenarios. For brevity, single characters are used instead of class names.

Table 1. Scenarios and Observed Footprints

Test Scenario	*Requirement*	*Observed Java Classes*
1. view textual movie information	[r2]	[C,E,J,N,R]
2. select/play movie	[r1][r6]	[A,C,D,F,G,I,J,K,N,O,T,R,U]
3. press stop button	[r8]	[A,C,D,F,G,I,K,O,T,U]
4. press play button	[r6][r9]	[A,C,D,F,G,I,K,N,O,T,R,U]
5. playing	[r0][r10][r12]	[A,C,D,F,G,I,K,O]
6. get textual movie information	[r5]	[N,R]
7. movie list	[r4]	[R]
8. VCR-like UI	[r7]	[A,C,D,F,G,I,K,N,O,R,T,U]
9. select movie	[r11]	[C,J,N,R,T,U]
10. press pause	[r3]	[A,C,D,F,G,I,K,O,U]

Our approach identifies trace dependencies between arbitrary requirements based on overlaps among the lines of code that implement those requirements. For example, there is a trace dependency between *scenario 4* (press play button) and *scenario 5* (playing) because the latter executes a subset of the lines of code the former does. Since both scenarios represent test cases for different requirements (*r9* and *r10*), we can infer a trace dependency between these requirements.

Figure 1 shows an excerpt of the VOD system. The figure contains links that visualize this subset/superset relationship among executed lines of code for requirements as discussed above. In particular, the requirement pointed to by the arrow uses a subset of the lines of code of the pointing requirement. For example, Figure 1 contains an arrow from *r9* to *r10* because *r9* uses a subset of the lines of code that *r10* does. There are also two clusters of requirements (e.g., *r1* and *r6*). Requirements in these clusters executed the exact same lines of code. Graphically this implies bi-directional trace dependencies among the requirements within those clusters which we simply abbreviate by forming clusters.

Figure 1 depicts some trace dependencies in green (solid lines) and others in red (dashed lines). Green lines imply correct trace dependencies while red lines represent incorrect trace dependencies. The latter occur because of two reasons: (1) sometimes the code of two requirements is interleaved such that the execution of the one always implies the execution of the other although both

[1] Note: Test scenarios can be defined in any form the developers desire since it is also their responsibility to test the system against those test scenarios. Automated techniques for capturing and testing test scenarios are readily available and thus outside the scope of this paper.

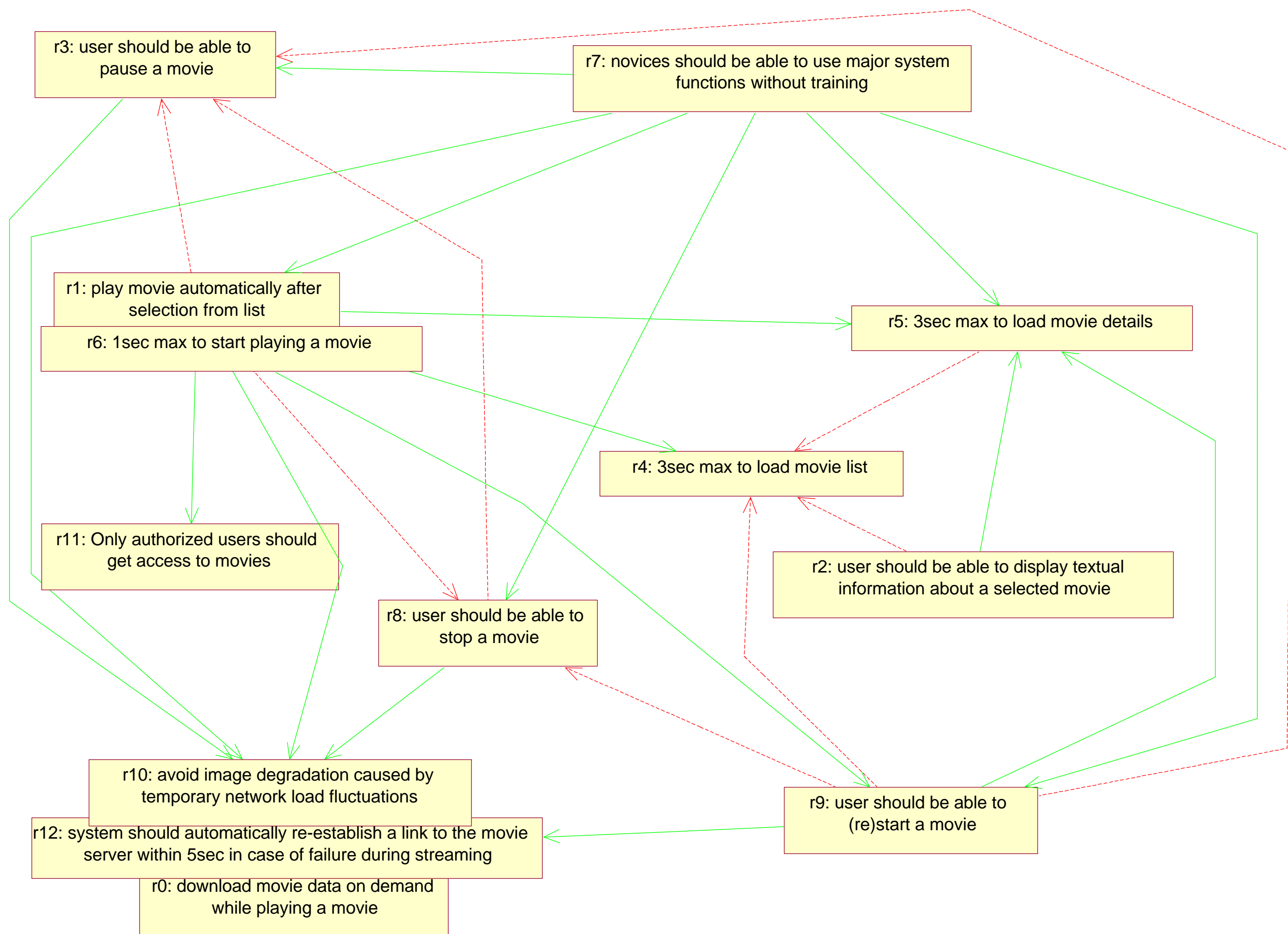

Figure 1: Automatically created trace links between VOD requirements.

requirements are indeed orthogonal (they to not interfere with one another) and (2) sometimes the granularity of the trace analysis is not fine enough. In our case, the latter is at fault. In order to keep the presented information small in this paper, we chose to use Java classes as the smallest entities. This is usually problematic since different requirements may well use the same java classes although different methods thereof. Indeed, if we were to do the trace analysis by comparing overlapping methods (instead of classes) then we would not have found any red links.

Nonetheless, incorrect trace results are bound to happen and it is important to understand the meaning of these trace dependencies to evaluate their correctness. The latter is important for identifying true dependencies for risk assessment and trade-of analyses. The figure shows that trace dependencies can get very complex even in this simple example confirming that there is a need for automation.

Given correct input, our approach is exhaustive in generating explicit trace dependencies for every artifact which results in a large number of (potentially incorrect) trace dependencies.

This has two implications:

1) what to do with these many trace dependencies?
2) how to deal with incorrect trace dependencies?

Thus far, we used our approach on a range of automated techniques for consistency checking and other forms of reasoning. In that context, problems generally arise because of the lack of information and not its abundance. The availability of abundant trace information provides more interconnectivity among modeling artifacts allowing deeper manual investigation. It is generally not necessary to "comprehend" the complete set of trace dependencies but only subsets addressing particular concerns.

As pointed out above, our approach also errs in producing incorrect trace dependencies at time. Here, our stance is that it is generally easier to dismiss incorrect trace dependencies, when encountered, instead discovering missing ones.

In the absence of a precise, complete, and automated approach to generating trace dependencies, we are faced with the decision of trading-off completeness and correctness. We believe our approach to be complete in

identifying all trace dependencies, however at the expense of also producing some incorrect ones. We found that our approach generally produces few incorrect trace dependencies compared to correct ones. Our approach is thus most useful in domains where completeness is desired (i.e., trade-off analysis, automation).

3. Understanding Trace Dependencies

Our approach identifies simple trace dependencies that do not convey any meaning or rationale. However, these simple dependencies are already very useful to support change management. For example, we know from the example in Section 2 that requirement *r9* depends on requirement *r10* and if requirement *r10* changes then requirement *r9* is affected by this change. Even in cases of incorrect trace dependencies (red link) this reasoning is useful since the change of one requirement may unknowingly result in the change of other, dependent requirements. However, while this reasoning about change management is very useful, we found that it is superficial in that it says little about how requirements affect one another.

3.1 Semantic Differences among Requirements

The trace dependencies derived through our approach are "dumb" links that merely express existing relationships. Human decision makers are thus required in clearly identifying the true nature of that relationship. However, upon investigating trace dependencies among requirements, we found that the meaning of these relationships is dependent on the types of requirements they bridge. For example:

- *Trace dependency between an efficiency requirement and a functional requirement.* The meaning of an efficiency (time behavior) requirement is to define some time constraint a (sub)system has to satisfy while the meaning of the functional requirement is to define user/customer requested functionality. If our approach identifies a "dumb" trace dependency between an efficiency requirement and a functional requirement (e.g., the dependency from *r6* to *r1*) then we may infer that the execution of this particular function has to satisfy the given performance constraint (e.g., the movies needs to be played within 1 seconds after selection from list). In doing so we thus identify an important property of the functional requirement going beyond mere change management.
- *Trace dependency between two efficiency requirements.* If our approach identifies a "dumb" trace dependency from an efficiency requirement (e.g., *r6*) to another efficiency requirement (e.g., *r5*) we may infer that the second requirement *r5* has to be at least as efficient as the first one (e.g., loading textual information about a movie has to be at least as fast as starting to play the movie). The reason is that if *r6* depends on *r5* then *r5* is executed as part of executing *r6*. Thus if *r6* has to finish executing with a given efficiency condition then, obviously, *r5* has to execute within the same or even better efficiency.

Table 2 defines these and additional strong implications.

Table 2: Strong Implication Table (Examples)

Dependency Type	*Implication*
efficiency *e* ? function *f*	Function *f* has to satisfy efficiency *e*
efficiency *e1* ? efficiency *e2*	*e2* has to be at least as efficient as *e1*
efficiency *r* ? security *s*	*s* needs to be realizable with efficiency *r*
Understandability *u* ? Recoverability *r*	the recovering action *r* should not contradict understandability
Function *f* ? Reliability *r*	*f* needs to be realizable within reliability *r*
Security *s* ? Function *f*	Function *f* must satisfy at least security *s*
Security *s1* ? Security *s2*	*s2* has to provide at least the level of *s1*

While Table 2 points out trace dependencies among requirements that have strong implications that are always true, there are also weaker cases listed in Table 3. For example, if a functional requirement depends on another functional requirement then an implication *may* be that the implementation of the second functionality is a pre-requisite for the implementation of the first one. In other words, eliminating the second requirement is useless if the first requirement is not eliminated either. Consider for example requirement *r1* "play movies automatically after selection from list" and, requirement *r0* "download movie data on demand from server while playing." Clearly, the second requirement needs to be implemented to support the first one.

The implications in Table 3 are weaker because functions may be orthogonal which means that two separate functionalities are implemented "close" to one other but in a way that their execution does not affect one another. For example, requirement *r1* "play movies automatically after selection from list" and requirement *r13* (not listed) "log the playing of movies in a log file". Clearly the second requirement is executed as part of the first requirement but the first requirement is indifferent to the second one.

Table 3: Weak Implication Table (Examples)

Dependency Type	*Implication*
function *f1* ? function *f2*	Implementing function *f2* might be a pre-requisite to implementing function *f1*
function *f* ? efficiency *p*	Some/most of the function *f* has to satisfy performance constraint *p*

Two pre-requisites are thus required for identifying meaningful strong and weak trace dependencies: (1) a classification of requirements (e.g, functional, efficiency, security), and (2) "dumb" trace dependencies among requirements. We demonstrated in previous work how we can identify "dumb" trace dependencies automatically as summarized in Section 2. Manual effort is still required for classifying requirements. As such, a single requirement may be classified into an arbitrary number of categories. Although a manual activity, we found that it is typically easy to categorize requirements this way. Indeed, looking at related work, we find that the classification of requirements is often a byproduct of existing requirements modeling techniques [8, 10].

3.2 Degrees of Overlaps

Our approach to finding trace dependencies relies on finding overlaps among test scenarios that belong to requirements (or groups of requirements). If two test scenarios for two different requirements overlap in the lines of code they execute (i.e., their "footprints" overlap) then we assume a trace dependency. In terms of identifying the implications of a trace dependency, our approach thus has a unique advantage. Depending on the degree of overlap in the lines of code executed we can strengthen and weaken the implications from Table 2 and Table 3.

Figure 2 shows how the lines of code of requirements may overlap. If there is no overlap (case 1) we can deduce that there is no trace dependency. In other words, if a security requirement affects different lines of code than a functional requirement then it is safe to say that the security requirement *does not* apply to the functional requirement. On the other extreme, if there is complete overlap (case 4) we can deduce that the requirements describe the same part of the system (e.g., clusters in Figure 1). In other words, if a security requirement is implemented by the same lines of code than a functional requirement then it must satisfy the security *exactly*; and that the security must be implemented by exactly this functionality (and no other). Both cases 1 and 4 in Figure 2 support strong reasoning and we can create reliable trace dependencies. However, case 4 is the least likely case. Typically, scenarios do not overlap in the lines of code they execute or they overlap only partially (cases 2 and 3). What does this imply?

If a requirement uses a subset of the lines of code (case 3) of another one Table 2 and Table 3 apply, but only in one direction. For example, if a functional requirement uses a subset of the lines of code of a security requirement then it must fully satisfy the latter; however, the security requirement will only be implemented partially by the functional requirement resulting in a reliable trace link in one direction and a weak one in the other direction. We found that many trace dependencies fall under this category. In Figure 1, all (correct) trace dependencies (green links) fall under this category.

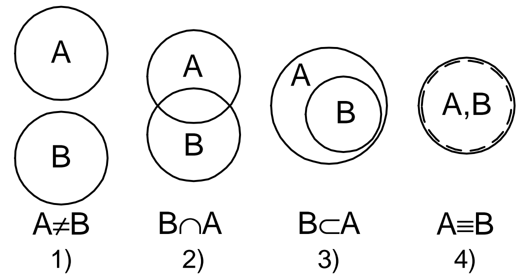

Figure 2. Types of Overlaps among Requirements

If a requirement overlaps with another requirement but both requirements have their unique source code (case 2) the implications we can derive from the trace dependency are weakened. For example, a security requirement that partially overlaps with a functional requirement implies that a part of the functionality has to implement the security (which part of the functionality remains unknown) and it implies that a part of the security is implemented by the functionality (again the part is unknown).

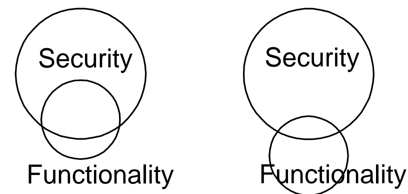

Figure 3: Partially overlapping requirements

Although this kind of dependency is weaker, it may still produce useful insights. Depending on how large the overlap is, we can gradually strengthen and weaken the meaning. If the functional requirement and the security requirement overlap 90% (almost complete overlap; left of Figure 3) then we can say that *most* of the functionality must satisfy the security constraint. Obviously, this is better than if the overlap is less (e.g., 20%; right of Figure 2). We can use this degree of overlap to distinguish between more and less reliable meaning.

For example, the efficiency requirements *r6* and the functional requirement *r2* partially overlap with the efficiency requirement using most of the same Java classes as the functional one. Due to the partial overlap, we cannot imply a strong meaning but because of the strong overlap, it is fair to say that most of *r2* has to have an efficiency of one second or less.

3.3 Grouping of Requirements

An interesting extension of this discussion is to consider packages of requirements instead of individual

requirements. This is not only important for the purpose of this paper but has relevance in real-world settings. For example, in requirements negotiation we need to understand the dependencies among requirements in order to allow meaningful trade-off analyses. It typically does not make sense to look at individual requirements – we usually have to build packages of related requirements in order to better handle complexity. In context of adding meaning to trace dependencies this "grouping" of requirements has further implications. Figure 4 shows that both *Security1* and *Security2* partially overlap with *Function1*. We cannot infer which part of the function has to satisfy *Security1* or *Security2*. But we see that a package of both security requirements together captures *all* of the functionality. Thus, we can deduce that all of Function1 *must* satisfy the weaker of the two security requirements; some of the functionality (and this part is unknown) has to satisfy the stronger of the two security requirements.

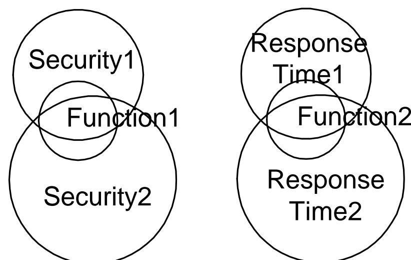

Figure 4: Grouping of Requirements

A similar example can be drafted with response time. If response times 1 and 2 overlap with a function then we can deduce that the response time of the functionality has to be less than the combined response times 1 and 2. It will be future work to investigate the effect of grouping requirements in more detail.

4. Deriving Positive and Negative Implications

As mentioned in the introduction, another possible real-world application for the approach discussed in this paper is architectural risk assessment. We intend to develop *decision support* to help engineers in identifying possible risks of adding new requirements or changing requirements of an existing system. The premise is that changes to requirements cause conflicts and that it is error-prone and expensive to investigate all trace dependencies manually in order to identify those conflicts. Our approach already provides strong support for this kind of trade-off analysis. If the trace analyzer *does not* find a trace dependency between two requirements then changing one is unlikely to cause conflicts with the other[2]. However, in what ways do requirement changes affect one another if there are trace dependencies?

Based on the meaning of dependencies, it is possible to determine a likelihood of risk. It is however incorrect to argue that the mere existence of requirements have negative or positive implications. For example, it seems intuitive that a security requirement has a negative impact on efficiency because security implies additional functionality and efficiency goes down with additional functionality. However, this reasoning is potentially wrong because it does not include information about the relative difference to some norm. For example, if the security requirement would define a functionality to be less than average secure then this would be good for efficiency.

It follows that we need information about the difference to some norm to reason about positive and negative implications of changes among requirements. A decision table could help us in determining these implications. Table 4 is a first attempt in defining such a decision support for trade-off analysis. For example, the table shows that a change in security (row) has a reverse effect on efficiency (column). As such, an increase in security has a high likelihood of a negative implication onto efficiency; and a reduction in security should have a positive implication. Other examples are: the addition of new functionality should also have a negative impact onto efficiency; the increase in security should have a positive impact onto reliability (i.e., we stipulate that a less secure system has to be more reliable for it to be more secure); or a change in security has an unknown impact onto understandability.

We believe that it is not necessary to define quality in terms of absolute numbers but only in terms of relative differences. Table 4 thus does not show a table with absolute rankings among qualities requirements. We envision such a table to be useful for a first-cut, trade-off analysis among requirements changes. As such, making a quality requirement stronger *does not* require investigating requirements without trace dependencies; it *likely does not* require investigating requirements with positive reinforcements; it *likely does* require investigating requirements with indifferent/unknown reinforcement; it *certainly does* require investigating requirements with negative reinforcement. As such, we believe that our technique can support trade-off analysis by guiding human decision makers and prioritizing their tasks.

5. Related Work

Different researchers have been developing approaches for modeling dependencies among quality requirements:

[2] Our approach clearly ignores potential side effects of data dependencies and emphasizes control dependencies only. We have not investigated this issue yet.

Table 4: Reinforcement among different quality requirements.

Child/Parent	Functionality	Efficiency	Understandability	Reliability	Security	Recoverability
Functionality	+	--	-	-	-	0
Efficiency	0	++	0	-	--	0
Understandability	-	0	++	+	0	+
Reliability	0	-	0	++	+	+
Security	0	-	0	+	++	+
Recoverability	0	-	+	+	+	++

Legend: ++ Strong positive reinforcement + Weak positive reinforcement 0 neutral - Weak negative reinforcement -- Strong negative reinforcement

Chung *et al.* discuss how architectural properties such as modifiability and performance can be modeled as "softgoals" and how different architectural designs support these goals. Architectural decisions can be traced back to stakeholders [2]. Franch and Maiden apply the *i** actor-based modeling approach [3] for modeling software architectures, not in terms of connectors and pipes, but in terms of actor dependencies to achieve goals, satisfy soft goals, use and consume resources, and undertake tasks [8].

Automated generation of traceability dependencies is also gaining increasing importance. For example, Zisman *et al.* present an approach for automatic generation and maintenance of traceability relations based on traceability rules. The artifacts as well as the rules are described in XML and supported by a prototype tool [15]. The approach has also been applied to organizational models specified in *i** and software systems models represented in UML [4].

6. Conclusion

The tables presented in this paper are initial work and certainly require more elaboration. However, we believe that the presented work is detailed enough to demonstrate the usefulness of defining the implications of trace dependencies based on the meaning of requirements. To date, we have demonstrated that we can determine trace dependencies among requirements automatically while observing usage scenarios. This requires either existing code (for execution) or hypotheses on the impact of scenarios onto code. And this requires hypotheses on the relationship between test scenarios and requirements.

We performed an initial experiment using the VOD system to determine how well the meaning of trace dependencies can be determined by the meaning of requirements. This only required the manual classification of requirements (i.e., grouping into qualities) which we found to be easy. As a result, we identified useful trace dependencies among system requirements that helped in identifying conflicts and risks. We also found that the meaning of trace dependencies can be strengthened and weakened depending on the degree of overlaps among requirements.

We believe that the discussed capabilities would ease the task of engineers in performing tasks such as assessing the risks of new or changing requirements.

7. References

[1] Boehm, B.W., Software Risk Management: Principles and Practices. *IEEE Software*, 1991. 8(1):32-41.

[2] Chung, L., Gross, D., and Yu, E., *Architectural Design to Meet Stakeholder Requirements*, in *Software Architecture*, Donohue, P., Editor. 1999, Kluwer Academic Publishers. p. 545-564.

[3] Chung, L., Nixon, B.A., Yu, E., and Mylopoulos, J., *Non-Functional Requirements in Software Engineering*. 2000: Kluwer.

[4] Cysneiros, F.G., Zisman, A., and Spanoudakis, G. A Traceability Approach for i* and UML Models. In: *Proceedings of 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems - ICSE 2003*. 2003.

[5] Dohyung, K., Java MPEG Player.

[6] Egyed, A. and Grünbacher, P. Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In: *17th Int'l Conf. Automated Software Engineering*. 2002. Edinburgh: IEEE CS.

[7] Egyed, A.F., A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering*, 2003. 29(2):116-132.

[8] Franch, X. and Maiden, N.A.M. Modeling Component Dependencies to Inform their Selection. In: *2nd International Conference on COTS-Based Software Systems*. 2003: Springer.

[9] Gotel, O. and Finkelstein, A. Contribution structures. In: *Second IEEE International Symposium on Requirements Engineering*. 1995. York, England.

[10] Grünbacher, P., Egyed, A., and Medvidovic, N. Reconciling Software Requirements and Architectures: The CBSP Approach. to appear: *Journal on Software and System Modeling* (SoSyM), Springer.

[11] ISO/IEC-9126, Information technology - Software Product Evaluation - Quality characteristics and guidelines for their use. 1991.

[12] Pohl, K., Brandenburg, M., and Gülich, A. Integrating Requirement and Architecture Information: A Scenario and Meta-Model Based Approach. In: *REFSQ Workshop*. 2001.

[13] Ramesh, B. and Jarke, M., Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 2001. 27(4):58-93.

[14] Ramesh, B., Stubbs, L.C., and Edwards, M., Lessons Learned from Implementing Requirements Traceability. *Crosstalk -- Journal of Defense Software Engineering*, 1995. 8(4):11-15.

[15] Zisman, A., Spanoudakis, G., Perez-Minana, E., and Krause, P. Tracing Software Requirements Artefacts. In: *The 2003 International Conference on Software Engineering Research and Practice (SERP'03)*. 2003. Las Vegas, Nevada, USA.