Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE),
Edinburgh, Scotland, UK, September 2002, pp. 163-171

Automating Requirements Traceability: Beyond the Record & Replay Paradigm

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way, Suite 231
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

Requirements traceability (RT) aims at defining relationships between stakeholder requirements and artifacts produced during the software development life-cycle. Although techniques for generating and validating RT are available, RT in practice often suffers from the enormous effort and complexity of creating and maintaining traces or from incomplete trace information that cannot assist engineers in real-world problems. In this paper we will present a tool-supported technique easing trace acquisition by generating trace information automatically. We will explain the approach using a video-on-demand system and show that the generated traces can be used in various engineering scenarios to solve RT-related problems.

1 Introduction

Requirements traceability (RT) is an approach defined as the "ability to describe and follow the life of a requirement, in both a forward and backward direction" [8] by defining and maintaining relationships to other artifacts created during system development [17] such as stakeholder needs, architectural or design elements, source code, to name but a few. Important goals of RT are to facilitate communication, to support integration of changes, to preserve design knowledge, to assure quality, and to prevent misunderstandings. RT is also crucial to establish and maintain consistency between heterogeneous models used throughout the system development life-cycle [13].

Recording and retrieving trace information can assist engineers to deal with important issues in system development or system maintenance [18]. For example, it could help to learn more about the origins of a requirement (e.g., the stakeholder needs) or about the rationale for a design choice. Engineers might also be interested in finding out how a functional or non-functional stakeholder need is realized in the system or if the implementation completely realizes the requirements. RT is especially important for analyzing the impact of new requirements or changes to existing ones.

The benefits of requirements traceability are widely accepted nowadays and sophisticated tool support is available to record, manage, and retrieve trace information [12]. However, several issues still hamper wide-scale adoption of RT in software engineering practice:

- Acquiring traces is still mostly a manual process with only little automation available. This results in enormous effort and complexity [16].
- The full potential of RT can only be exploited if complete trace information is available. However, missing information is a reality.
- RT are in a constant state of flux since they may change whenever requirements or other development artifacts change (e.g., evolution).
- It is often hard to anticipate the kind of engineering issues that might arise later. The available recorded trace information might therefore be insufficient.
- Traces have to be identified and recorded between a high number and heterogeneous set of engineering artifacts (document, models, code, …). It very challenging to create meaningful relationships in such a complex context.

We believe that automating RT should go beyond recording and replaying trace information. In this paper we will present a technique that can ease trace acquisition by generating trace information automatically. It must be noted that this work is a continuation of our earlier work on identifying trace dependencies using scenarios [7]. The contribution of this paper is on showing (1) how requirements must be defined as input, (2) how RT results derived by our approach have to be interpreted, (3) how functional and non-functional requirements are treated, (4) how useful the derived RT results are in addressing requirements-related problems, and (5) how evolutionary/incremental requirements engineering is supported.

The remainder of this paper is organized as follows: In Section 2 we briefly explain the Trace Analyzer technique. Section 3 will introduce the Video-On-Demand (VOD) case study. Section 4 will demonstrate that the generated traces are useful to support real-world engineering scenarios. In section 5 we will discuss related work. Conclusions and an outlook on further work will round out the paper.

2 Automating RT using Trace Analyzer

Trace dependencies describe origin, rationale, or realization of software development artifacts. For instance, if a requirement *R* led to the implementation of some source code *C* then there is a trace dependency between the two. If the requirement changes then the source code is affected. Conversely, if the source code changes then the requirement is affected. Bi-directionality is very important for trace analysis and implies that if *R* depends on *C* then *C* depends on, at least, *R*.

Based on this simple property, the Trace Analyzer [7] defines trace dependencies through (a) transitive reasoning and (b) shared use of a "common ground." Transitivity is an intrinsic property of trace dependencies. It defines A to depend on C if A depends on B and B depends on C. Shared use of a common ground is a more subtle but very powerful form of deriving trace dependencies. To use it, one has to identify a "common ground" with the following property: Given that *A* and *B* depend on subsets of that common ground then a trace dependency exists if and only if those subsets overlap. We found the source code of a software system to be a powerful candidate for a common ground. If requirement *A* depends on some source code C_A and requirement *B* also depends on some source code C_B then one can infer that *A* and *B* depend on one another if C_A and C_B overlap. The rationale for this can be inferred from bi-directionality.

The Trace Analyzer technique takes known or hypothesized dependencies between software development artifacts (e.g., requirements) and common ground (e.g., source code). It then builds a graph consisting of nodes that contain those common grounds and all their overlaps (e.g., separate nodes for C_A and C_B but if they overlap then this is captured explicitly in yet another node). This graph is then subjected to various manipulations to move known artifacts between the nodes. The goal is to constrain for all nodes in the graph what artifacts they relate to and what artifacts they do not relate to. Trace analysis is complicated by imprecise input where single dependencies may include multiple artifacts (*A* or *B* depends on *C*) and trace analysis is complicated by open-ended input where only partial knowledge is available (*A* depends on *C* and possibly others).

Trace analysis is an iterative process using a large number of rules to manipulate the graph structure. In a final step, the graph is traversed once more to identify all nodes related to individual artifacts. Trace dependencies are then established if two different artifacts relate to at least one common node. The graph may even help in determining the "strength" of a dependency based on the number of nodes any two artifacts have in common.

The trace analyzer technique is fully automatable and tool supported. The only deficiency, as it may appear, is that some trace dependency input has to be provided manually. However, even this input can be generated (semi-)automatically if either an executable software system exists (e.g., source code) or its model can be simulated. We use test scenarios to define how to test individual artifacts or groups of artifacts. While testing a scenario on a real system, it can then be observed what implementation classes, methods, and lines of code are used. For instance, we employ the commercial tool Rational Pure Coverage® to observe test scenarios on an executing system. With the help of such a tool, trace dependencies between test scenarios and source code can be generated automatically. Given that we know what artifacts a test scenario relates to (the premise) one can automatically infer trace dependencies between artifacts and code using transitive reasoning. These trace dependencies are then used as input to the trace analyzer to provide full automation.

3 VOD Case Study

This paper will highlight the benefits of using the Trace Analyzer technique [7] for automating RT. The discussion in this paper will be supported by the case study of a video-on-demand (VOD) system, which was developed by a third party [5]. This software system is essentially a movie player that can search for movies, select and play them. The "on-demand" feature of the

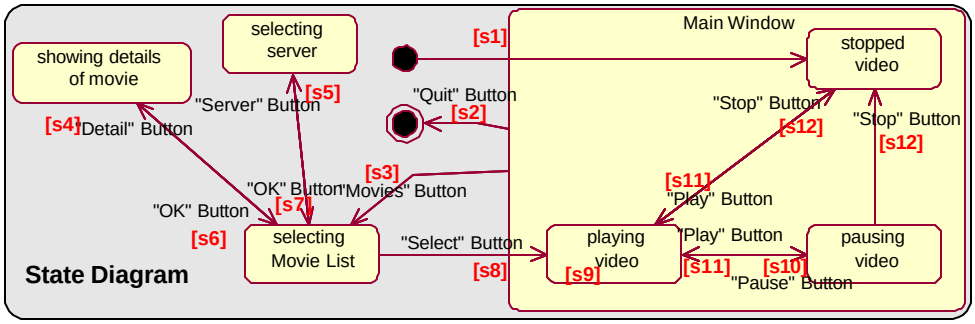

Figure 1: Statechart Diagram

player allows the playing of a movie concurrently while downloading its data from a remote site. This system provides an interesting challenge for Requirements Traceability because its complex computational logic is well-hidden underneath a simple VCR-like user interface (play, pause, stop button). Both functional and non-functional issues are fundamental in defining the requirements of the VOD system.

3.1 VOD System and Model

The VOD system consists of 21 Java application-specific classes, and uses a large number of off-the-shelf library classes. The VOD system was also modeled using various UML diagrams (see statechart diagram in Figure 1 as one example). Requirements were not defined but for the purpose of extending the VOD system (discussed later), the authors re-engineered those requirements manually. Table 1 depicts an excerpt of the captured requirements. For instance, requirement r7 defines the need for an intuitive user interface modeled after a VCR player. Requirement r6 defines a maximum delay of one second to start playing a movie once it has been selected.

Table 1. List of Requirements

r0	Users should be able to display a list of available movies and select one from the list
r1	Play movie immediately after selection from list
r2	Users should be able to display textual information about a selected movie
r3	User should be able to pause a movie
r4	3 seconds max to load movie list
r5	3 seconds max to load textual information about a movie
r6	1 second max to start playing a movie
r7	Provide VCR-like user interface
r8	User should be able to stop a movie
r9	User should be able to start a movie

Figure 1 shows a state diagram of the VOD system describing its behavior. It depicts that the VOD system operates either in a movie selection mode (left) or in a movie playing mode (right). During movie selection, a user can select servers for downloading movie lists, inspect textual information about movies, and select individual movies for playing. During playing mode, a selected movie may be paused, stopped, and played again. The transitions between these states correspond to buttons a user may press in the VOD's user interface. For instance, a user may press the "Movies" Button at any time during movie playing to select another movie.

The drawback of the existing requirements and models (diagrams) is that no trace dependencies are defined between them. In some cases, those trace dependencies could be guessed relatively easy but the informal capture of the requirements and the semi-formal nature of the UML models make it hard to identify complete and correct trace dependencies manually. The following will show how our scenario-based trace analysis approach can automatically define trace dependencies among requirements, between requirements and code, and between requirements and model elements (e.g., state transitions).

3.2 Scenarios and Observations

In order to identify trace dependencies fully automatically our approach requires the existence of usage scenarios that can be tested against the code. Table 2 lists all test scenarios defined for this case study. For example, test scenario 1 uses the VOD system to display a list of movies. The details of how to test this scenario on the system are omitted here for brevity but the test scenario describes how to configure the VOD system and what user interface actions to perform (e.g., press buttons) in order to achieve the desired results. We then used the commercial tool Rational PureCoverage to monitor the VOD system while testing the scenario. In the course of testing scenario 1, we observed that only four Java classes got executed: BorderPanel (C), ListFrame (J), ServerReq (R), and VODClient (U). In the following, we will only use single letter acronyms for Java classes. The letters C, J, R, and U are associated with the Java classes mentioned above.

Table 2 also shows what artifacts (model element, requirements) the test scenarios apply to. For instance, test scenario 1 was defined to relate to the state transition [s3] "Movies Button" in the statechart diagram (see Figure 1). This implies that test scenario 1 is a test case for the state transition [s3] and, while executing it on the real system, it was observed to use the Java classes (code) [C,J,R,U]. Due to transitivity of trace dependencies, one may conclude that the state transition [s03] depends on the code [C,J,R,U].

Note that if non-observable components (COTS, libraries, …) are used then often their "wrappers" or "glue code" can be used as substitute if necessary.

Table 2. Scenarios and Observed Footprints

Test Scenario	*Artifact*	*Observed Java Classes*
1. view movie list	[s3]	[C,J,R,U]
2. view textual movie information	[s4,s6][r2]	[C,E,J,N,R]
3. select/play movie	[s8,s9][r6]	[A,C,D,F,G,I,J,K,N,O,T,R,U]
4. press stop button	[s9,s12][r8]	[A,C,D,F,G,I,K,O,T,U]
5. press play button	[s9,s11][r9]	[A,C,D,F,G,I,K,N,O,T,R,U]
6. change server	[s5,s7]	[C,R,J,S]
7. playing	[s9]	[A,C,D,F,G,I,K,O]
8. get textual movie information	[r5]	[N,R]
9. movie list	[r4]	[R]
10. VCR-like UI	[r7]	[A,C,D,F,G,I,K,N,O,R,T,U]
11. select movie	[r0]	[C,J,N,R,T,U]
12. select/play movie	[r1]	[A,C,D,F,G,I,J,K,N,O,R,T,U]
13. press pause	[s9,s10][r3]	[A,C,D,F,G,I,K,O,U]

Table 2 defines 12 additional scenarios. This includes one test scenario for each requirement (although multiple may exist) and, to make the trace analysis more interesting, this also includes some ambiguous test cases for the statechart diagram. A trace dependency is ambiguous if it does not precisely define relationships between artifacts. For instance, test scenario 2 defines the state transitions [s4] and [s6] to relate to the code [C,E,J,N,R]. This statement is ambiguous in that it is unclear which subset of [C,E,J,N,R] actually belongs to [s4] and which subset belongs to [s6].

The approach relies on the capability of a software engineer to relate the test scenarios to the requirements and to the model elements. Three errors are possible that may impact the trace analysis in different ways: (1) the engineer omits a link between a test scenario and a requirement, (2) the engineer creates a wrong link, or (3) there is a mismatch between a requirement and the specified tests (for example, the test case only exercise the wrong or only a partial functionality). Although the technique has some means of detecting inconsistencies among links it can be fooled this way and engineers need to be aware of this.

3.3 Trace Analysis

Scenario-based trace analysis is simple for test scenarios that are unambiguous. For instance, the requirement [r6] defines a maximum delay of 1 second to start playing a movie. We know from test scenario 3 that [r6] was observed to execute the Java classes [A,C,D,F,G,I,K,O]. Consequently, this Java code needs to be optimized to perform as desired.

The trace analysis is complicated by the use of ambiguous scenarios. For instance, through scenario 5 we know that pressing the play button causes [s11] directly and [s9] (playing the actually movie) indirectly. Altogether [s9,s11] use 13 Java classes [A,C,D,F,G,I,K,N,O,R,T,U] but it is left unspecified which subsets of those classes are used by [s11] or [s9]. Alternatively, through scenario 7 we learn that [s9] alone uses the Java classes [A,C,D,F,G,I,K,O] which is a subset of [s9,s11]. It is thus possible to reason that classes [A,C,D,F,G,I,K,O] belong to [s9] whereas the remaining classes [N,R,T,U] belong to [s11].

For reasons of efficiency and precision, our trace analyzer approach uses a graph structure (called the footprint graph) to infer trace dependencies. Footprints are the observed lines of code executed while testing scenarios. Figure 2 shows a partial footprint graph based on scenarios 1, 4, 5, 7, and 13. Lower nodes (child nodes) are subsets of higher nodes (parent nodes). This subset relationship applies to both the model elements and Java classes used. For instance, scenario 7 about playing a movie [s9] uses a subset of the lines of code that scenario 5 uses. Scenario 7 [s9] furthermore refers to a subset of the model elements referred to by Scenario 5 [s9,s11]. Within the footprint graph this places the node for Scenario 7 below the node for Scenario 5.

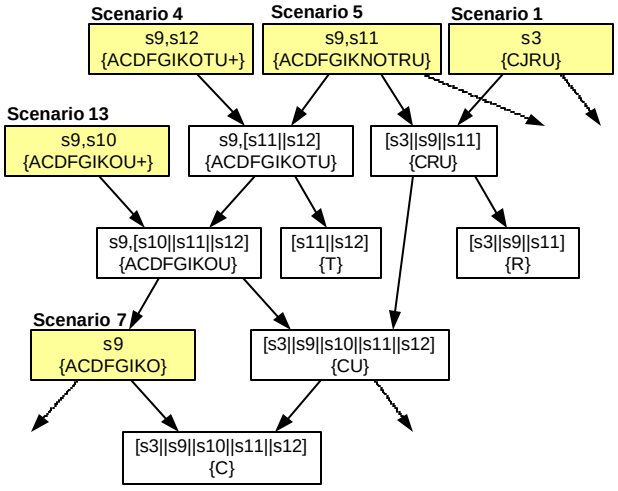

Figure 2. Partial Footprint Graph

Besides containing a node for every scenario, the footprint graph also contains nodes for all possible overlaps between those scenarios. For instance, scenario 1 overlaps with scenario 5 in their common use of the Java classes [C,R,U]. A node, child to both, was thus introduced to capture that overlap explicitly. An irregularity is Scenario 4 which uses a subset of the Java classes that Scenario 5 uses but in the graph figure it is not a child node to Scenario 5 because Scenario 4 [s9,s12] does not refer to a subset of the model elements that Scenario 5 refers to [s9,s11]. It is thus presumed that both scenarios use similar lines of code but are different in nature. Thus an explicit node is added to the graph to avoid having to add the one as a child of the other.

Once all scenarios are inserted into the footprint graph, the graph contains nodes for every possible overlap between any two scenarios. The graph is then manipulated to move artifacts around those nodes to identify for every node the artifacts it could possibly relate to. For instance, parent node [s9,s11] has two children. Each child relates to a subset of the Java classes of the parent but both children together relate to the same Java classes the parent does. For consistency and completeness, both children of parent node [s9,s11] together must thus relate to [s9] and [s11] but each child individually may relate to a subset of [s9,s11]. In case of the left child, we know that it must relate to [s9] given that both parents of that child have [s9] in common. In case of the right child, we do not yet know what artifacts apply and we thus presume a subset of the parent artifacts. This uncertainty is captured in form of an ambiguity in that either [s3], [s9], [s11] or all of them relate to the right child node (indicated as [s3||s9||s11]. Similarly, the left child may also relate to

[s11] and/or [s12] since we are uncertain about those elements as well.

The trace analyzer technique uses a larger set of rules than can be described in this paper. There are many special cases one has to consider to make the trace analysis reliable (see [7] for a detailed discussion). All rules have in common that they move model elements within the graph structure with the goal of identifying for every node what model elements they relate to. Since the leaf nodes (nodes without children) refer to individual Java classes (in this case study only since leaf nodes may also refer to Java methods or individual lines of code), it is then possible to infer what model elements a Java class relates to. Table 3 below summarizes some dependencies between artifacts and code that can be interpreted from the graph.

Table 3. Artifact to Java class dependencies

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	O	R	S	T	U
r0			F							F				F				F		F	F
r1	F		F	F		F	F		F	F	F			F	F			F		F	F
r2			F		F					F				F				F			
r3	F		F	F		F	F		F		F				F						F
r4																		F			
r5														F				F			
r6	F		F	F		F	F		F	F	F			F	F			F		F	F
r7	F		F	F		F	F		F		F			F	F			F		F	F
r8	F		F	F		F	F		F		F				F					F	F
r9	F		F	F		F	F		F		F			F	F			F		F	F
s3			P							P								P			P
s9	F		P	F		F	F		F		F				F						
s10			P																		P
s11			P											P				P		P	P
s12			P																	P	P

For instance, we can interpret from the footprint graph that either model element [s11] or [s12] or both of them have a dependency to the Java class 'T.' Table 3 thus shows this dependency in form of a letter where the column 'T' and the rows 's11' and 's12' intersect. The letter indicates the confidence of the trace analyzer in this finding: 'F' for full confidence; 'P' for partial confidence. The trace analyzer determined that class 'T' either depends on s11 or s12. Consequently one may only have a partial confidence that s11 depends on class 'T.' In fact, one may only then conclude that s11 depends on class 'T' if it becomes known that s12 does not depend on class 'T.'

In some cases, the trace analyzer technique can reduce ambiguous inputs. For instance, scenario 3 in Table 2 defined [s8] to potentially depend on class 'F' however the trace analyzer concluded that class 'F' belongs to [s9] instead. Although the trace analyzer technique can reduce ambiguity, it cannot necessarily eliminate it in all cases. Ambiguous dependencies are the result of ambiguous and/or incomplete input. The more precise the input the less ambiguous the dependencies. The trace analyzer can also identify some forms of inconsistent input but its discussion is out of the scope of this paper.

3.4 Interpreting the graph

Table 3 only captures trace dependencies between requirements and code, and between model elements and code. Given the transitive property of trace dependencies, one can also use Table 3 and the footprint graph in Figure 2 to infer dependencies between different requirements and/or model elements. For instance, it was determined that r9 traces to [A,C,D,F,G,I,K,N,O,R,T,U] and that [r6] traces to [A,C,D,F,G,I,J,K,N,O,R,T,U]. Given that [r9] traces to a subset of the classes that [r6] traces to implies a dependency between [r9] an [r6]. In other words, the requirement for a 'play button' also implies the non-functional constraint of only having 1 second to start playing the movie once the button is pressed.

Whereas some dependencies are intuitive and could be determined manually with relative little effort, the following dependency is not only hard to guess but also problematic: One can observe through Table 3 that [r6] depends on [r5] given that [r5] traces to the Java classes [N,R] (a subset of [A,C,D,F,G,I,J,K,N,O,R,T,U]). This dependency implies that in order to start playing a movie one needs to load the textual information about a movie. The problem is that loading this information is allowed to take up to three seconds which is longer that the allowed 1 seconds max to start playing that movie. The finding of this trace dependency implies a conflict between two requirements.

Besides finding trace dependencies between different requirements, the trace analyzer technique also finds dependencies between requirements and model elements. For instance, the requirement [r0] "display and select movie from list" depends on the state transition [s3] because [s3] may at most relate to [C,J,R,U] whereas [r0] is known to relate to [C,J,N,R,U] (a superset). Using the same method, one may identify many more trace dependencies.

3.5 Determine impact of new or changed requirements

The trace analyzer technique can also help in analyzing the impact of new or changed requirements, which are common in an iterative software process [4]. Normally, it is desirable to validate a new requirement or a change in an existing requirement prior to implementation. In such a case where no common ground (e.g., source code) exists one can still use the trace analyzer by hypothesizing about the impact of a new requirement or a requirement change. The following discusses one such case in context of adding a new requirements to the VOD system that has not yet been implemented.

Table 4. New/Changed Requirements

r6	3 second max to start playing a movie
r10	Avoid image degradation caused by temporary network-load fluctuations

Table 4 shows the requirement [r6] changed due to the conflict with [r9] and it also shows a new requirement that deals with image degradation because of network fluctuations. Recall that the VOD system is a video-on-demand system that starts playing a movie as soon as data arrives via the network. If a temporary network congestion causes delays it may negatively affect image quality. A possible approach to achieve requirement [r10] would be to do some initial caching to overcome this limitation. To find out whether this new requirement clashes with some existing requirements, we can do a preliminary trace analysis. To do this, we hypothesize about the impact of the new requirement. Thus, we presume that caching can be done solely by modifying the Java classes [A,D,G,I,K,O] plus adding some new ones. We thus define a new, hypothetical trace dependency between [r10] and [A,D,G,I,K,O,+] and repeat the trace analysis with this additional data.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
r10	F			F			F		F		F				F						

If the new hypothesized trace dependency is compared with the other known trace dependencies in Table 3 then we can again determine trace dependencies between [r10] and other artifacts based on their overlapping use of common code. For instance, one can tell that the new requirement [r10] uses a subset of the code that the state [s9] uses. Consequently, [s9] fully depends on [r10] (100% strength). Even more interesting is that the changed requirement [r6] also fully depends on [r10].

Recall that [r6] was changed because 1-second response time was insufficient given that at least 3 seconds are needed to load textual information about a movie. We thus relaxed the 1-second constraint to three seconds. However, now we learn that if requirement [r10] should get implemented then this will cause additional delays. A partial caching of a movie can only happen once movie details are known. A caching period of 1-2 seconds thus adds to the already three seconds needed to load and play a movie. This is a conflict that can be identified with ease once one is aware of the trace dependency. This example again shows that our trace analyzer approach can help in pinpointing non-obvious dependencies between artifacts. If those dependencies lead to the identification of conflicts then the trace analyzer can further help in evaluating potential solutions.

4 Benefits of the Approach for Requirements Traceability

Requirements Traceability is an important means to facilitate communication among the success-critical stakeholders, to ease determining the impact of changes and support their integration, to preserve knowledge and dependencies created during the design process, to assure quality, and to prevent misunderstandings. This section will discuss how our automated approach towards deriving trace dependencies can assist engineers in dealing with real-world scenarios.

Understanding requirements' origins and rationale. Traceability between stakeholder needs and requirements can be detected manually but our automated technique provides a more complete traceability. Trace analysis can derive missing relationships between informal user needs and existing design elements. For example, the automated technique can help to create traceability links from new stakeholder needs to existing design: In one of our experiments a link from the new user need "Users should be able to capture movie screen snapshot at any time" to the "Pause button" GUI element was automatically derived. This gives rationale and explains why an element is here by providing backward traceability.

Traceability to non-functional requirements. Using the approach even non-functional requirements can be linked to model elements or code sections. Non-functional stakeholder needs ultimately always result in some code although this relationship is typically almost impossible to identify. For example, we know that implementation class [U] exists because of [r7] and [r8]. The requirements "Three seconds max to load textual information about a movie" can be linked to implementation classes [N,R]. By generating missing trace information the trace analyzer technique can link a new non-functional user need "Novices should be able to use the most important functions without training" to requirements r1, r3, r8, r9 and thereby also show all affected implementation classes.

Aid identification of conflicting requirements. It is typically hard to derive all dependent requirements because of scalability issues. An automated approach towards generating dependencies between requirements is thus critical to determine whether dependent requirements are consistent. For example, the requirement "novices should be able to use system without training" may be in conflict with the requirement "3 seconds response time" because such a long delay would not be intuitive. Our approach cannot automatically derive conflicts, but by finding all possible dependencies it is easier to identify potential inconsistencies and conflicts. Another example is discussed in Section 3.4.

Verification of requirements. An important task of a software engineer is to determine whether the requirements have been realized properly. We specify acceptance test through scenarios for all requirements. We can thereby make sure that scenarios sufficiently cover requirements. The case study shows that we tested all requirements and know what sections of the code realize them.

Identification of missing requirements. The approach can also be used to identify missing requirements. For example, the analysis reveals that we have no requirements defining scenarios for 'S' or 'P'. Does this mean that the system implements something not stated in the requirements? Through the generated trace dependencies we know that implementation class 'S' is about [s5] and [s7] (selecting server) and we can now reason that no requirement was defined that allows the user to change servers. Besides detecting missing requirements, we can also reason about missing or incomplete designs. For instance, we find that design element [s1] was not defined in any requirement or any implementation.

Determination of change impact. Assume that the response time in requirements "3 second max to start playing a movie" has to be reduced. Trace analysis reveals the impact of such a change onto other requirements, onto design, and onto code. But trace analysis also reveals the reverse impacts. For example, we know without any manual creation of trace information that if code element [T] (Video.java) changes than the design elements [s11,s12] are affected by that change.

Determination of impact of new requirements. New requirements are an interesting case for deriving trace dependencies where parts of the system have not even been built. Section 3.5 discussed that by hypothesizing what model elements/code might be affected by a new requirements the trace analyzer can predict which requirements and other development artifacts might be affected.

Understanding level of strength of dependencies. It must be noted that our technique can determine strength of dependencies where strength is defined in terms of how many classes (or methods or lines of code) two artifacts have in common. For instance, it can be observed that that [r3] uses 33% of the classes of [r0] and [r0] uses 22% of the classes of [r3] (the percentage applies to the number of overlapping classes versus total classes). Although the dependency between r3 and r0 is not very strong it still implies that a change in [r3] has a 33% chance that it will also affect [r0]. This percentage of course presumes that all classes are of equal size which they are not. For more precise dependency numbers, the trace analysis could be conducted on methods or lines of code. Note that the strength of a dependency is not to be confused with the confidence in a dependency. Whereas the confidence (full/partial) defines the likelihood of false positives, strength simply describes the degree of overlaps.

Upon inspection of the generated traces between requirements, we find that most requirements trace to most other requirements at least partially. This is not very surprising since requirements tend to be very generic descriptions. For a more useful determination of trace dependencies between requirements one should focus more on the extremes – that is 0% and 100%. If there is 0% overlap between two requirements then there is no dependency between them. The requirement [r3] (pause movie) has nothing in common with requirement [r4] (Three secs max to load movie list). If there is a 100% overlap between two requirements then there is a strong dependency between them. For instance, [r6] uses 100% of [r5] which implies a strong dependency.

Domain specific code vs generic code. The approach can also be used to distinguish project or domain-specific code from generic code. For specific analyses it might be necessary to ignore generic code since it is likely to be used for different purposes and may obscure analysis. For instance, two classes may use a common third class to create and modify a file but this does not mean those two classes are related to one another (note: those two classes may be related if they modify the same file but the trace analyzer approach cannot detect that).

Determining artifacts needing attention. Special care has to be spent on very complex and/or very important artifacts. The importance of a development artifact depends on how many other artifacts it constrains (e.g., design element s9 is important in that it (partially) defines 8 implementation classes). The complexity of an artifact depends on how many other artifacts constrain it (e.g., design element s9 is also complex since 6 out of the 10 requirements impose themselves on it). Trace analysis can simply pinpoint these kinds of metrics, e.g., for complexity versus importance trace offs.

Balancing granularity of requirements. In an early project stage requirements will be typically fairly generic; later on requirements will be more specific – unless, of course, a major change comes along. For instance, [r5] is a lower-level requirement than [r6] because [r5] uses a subset of the code than [r6] does. Trace analyzer can find requirements that are very generic (e.g., they affect many classes). This can assist the engineer to balance out requirements by increasing precision.

5 Related Work

Different approaches have been developed to automate the acquisition of trace information. Typically these approaches support the creation or recovery of traces between two types of engineering artifacts (e.g., de-

sign/code, code/documentation, requirements/architectures).

For example, Antoniol et al. discuss a technique for automatically recovering traceability links between object-oriented design models and code based on determining the similarity of paired elements from design and code [2]. Murphy et al [13] suggest software reflexion model showing where the engineer's high-level model agrees with and where it differs from a model of the source. Antoniol et al describe an approach to automatically recovering trace information between code and documentation [1].

Other approaches discuss specific traceability issues without focusing on automation: Arlow et al emphasize the need to establish and maintain traceability between requirements and UML design and present Literate Modeling as an approach to ease this task [3]. Gotel and Finkelstein extend the view of artifact based RT and focus on understanding the social network of people that contributed in the development of requirements [9]. Pohl et al describe an approach based on scenarios and meta-models to bridge requirements and architectures [15]. Grünbacher et al. discuss the CBSP approach that improves traceability between informal requirements and architectural models by developing an intermediate model based on architectural dimensions [11].

Gruber *et al.* discuss the problems of design rationale capture and demand the need for automatically inferring rationale information [10].

6 Conclusions and Further Work

In this paper we presented an approach supporting the automated generation of trace information. We discuss the approach in the context of a video-on-demand system and show that it allows deriving trace dependencies between the different models and artifacts of the system automatically. We then discussed how the derived traces can support engineers. A major strength of the approach is that it creates many non-obvious dependencies allowing more thorough reasoning and pinpointing of non-standard situations.

A key contribution of our approach is that it reduces the enormous effort and complexity of acquiring traces by automatically deriving trace information from a small set of obvious hypothesized traces. This leads to more complete traces and the full potential of RT can be exploited: For example, traces to pre-requirements explaining where requirements come from or traces from/to non-functional requirements are typically difficult to create and maintain using manual traditional approaches. The automated approach also creates traces engineers typically could not anticipate. This improves the applicability of our approach in different contexts or non-standard engineering problems.

Further work will concentrate on developing automated support assisting engineers in exploring and using the automatically derived trace dependencies. For example, by highlighting artifacts and situations that require special attention [5]. Another thread of our research will focus on experimenting with different levels of granularity of coverage measurement: The technique allows to specify this level arbitrarily (e.g., class, method, or statement). We aim at developing heuristics allowing software engineers to determine the optimum level of granularity in a given situation. We are also intending to apply our technique and findings to a large-scale system.

References

[1] Antoniol, G., Canfora, G., De Lucia, A., Casazza, G. Information Retrieval Models for Recovering Traceability Links between Code and Documentation Proceedings of the International Conference on Software Maintenance, 2000.

[2] Antoniol, G., Caprile, B., Potrich, A., Tonella, P., Design-Code Traceability Recovery: Selecting the Basic Linkage Properties, Science of Computer Programming, vol. 40, issue 2-3, pp. 213-234, July 2001.

[3] Arlow, J., Emmerich, W., Quinn, J., Literate Modelling - Capturing Business Knowledge with the UML, UML'98: Beyond the Notation 1998.

[4] Boehm B., Egyed A., Kwan J., Port D., Shah A., Madachy R., Using the WinWin Spiral Model: A Case Study, IEEE Computer, 7:33–44, 1998.

[5] Dohyung, K., "Java MPEG Player," Online at http://mirage.snu.ac.kr/dhkim/java/MPEG/.

[6] Egyed A., Boehm B., Comparing Software System Requirements Negotiation Patterns. *Systems Engineering Journal*, Vol.6/1:1-14, 1999.

[7] Egyed, A., A Scenario-Driven Approach to Traceability, Proceedings of the 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001, pp. 123-132.

[8] Gotel O.C.Z., Finkelstein A.C.W., An Analysis of the Requirements Traceability Problem. *1st International Conference on Rqts. Eng.*, pp. 94-101, 1994.

[9] Gotel, O. & Finkelstein, A. "Extended Requirements Traceability: results of an industrial case study" in Proc. 3rd International Symposium on Requirements Engineering RE97, (IEEE CS Press), 1997, 169-178.

[10] Gruber, T. R. & Russell, D. M. Generative design rationale, Lawrence Erlbaum Associates, 1994.

[11] Grünbacher P., Egyed A., Medvidovic N., Reconciling Software Requirements and Architectures: The CBSP Approach, In: Proceedings 5th IEEE International Symposium on Requirements Engineering (RE01), Toronto, Canada, 2001.

[12] INCOSE requirements management tool survey, Online at http://www.incose.org

[13] Medvidovic N., Grünbacher P., Egyed A., Boehm B.W., Bridging Models across the Software Lifecycle, *to appear:* Journal of Systems and Software, 2002.

[14] Murphy, G. C., Notkin, D. and Sullivan, K., Software Reflexion Models: Bridging the Gap Between Source and High-Level Models, In the Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, October 1995, ACM, New York, NY, p. 18-28.

[15] Pohl et al., Integrating Requirement and Architecture Information: A Scenario and Meta-Model Based Approach, REFSQ workshop 2001.

[16] Ramesh, Bala; Stubbs, Lt Curtis; & Edwards, Michael. "Lessons Learned from Implementing Requirements Traceability." Crosstalk, Journal of Defense Software Engineering 8, 4 (April 1995): 11-15. Online at http://www.stsc.hill.af.mil/crosstalk/1995/apr/Lessons.asp

[17] Ramesh, B., Jarke, M., Toward Reference Models for Requirements Traceability, IEEE Transactions On Software Engineering, vol. 27, no. 1, January 2001.

[18] Software Productivity Solutions, Analysis of Automated Requirements Management Capabilities. Melbourne, FL:, 1994.