International Journal of Software Engineering and Knowledge Engineering
Vol. 0, No. 0 (1994) 000–000

SUPPORTING SOFTWARE UNDERSTANDING WITH AUTOMATED REQUIREMENTS TRACEABILITY

ALEXANDER EGYED
Teknowledge Corporation, 4640 Admiralty Way, Suite 1010
Marina Del Rey, CA 90292, USA

and

PAUL GRÜNBACHER
Systems Engineering and Automation, Johannes Kepler University
4040 Linz, Austria

Received (June 7, 2004)
Revised (January 27, 2005)
Accepted (accepted date)

Requirements traceability (RT) aims at defining and utilizing relationships between stakeholder requirements and artifacts produced during the software development life-cycle and provides an important means to foster software understanding. 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. This results in invalid or incomplete trace information which cannot support engineers in real-world problems. In this paper we present a tool-supported approach that requires the designer to specify some trace dependencies but eases trace acquisition by generating others automatically. We illustrate the approach using a video-on-demand system and show how the generated traces can be used in various engineering scenarios to improve software understanding. In a case study using an open source software application we demonstrate that the approach is capable of dealing with large-scale problems and delivers valid results.

Keywords: Software understanding, software traceability, automation

1. Introduction

Requirements traceability (RT) is defined as the "ability to describe and follow the life of a requirement, in both a forward and backward direction" [1] by defining and maintaining relationships to related development artifacts [2] such as stakeholder needs, architectural components, design model elements, or source code. RT is considered crucial for establishing and maintaining consistency between heterogeneous models used throughout the development life-cycle [3]. Frequently reported benefits of RT include the facilitation of communication, support for the integration of changes, the preservation of design knowledge, quality assurance, the prevention of misunderstandings. Trace information fosters software understanding and assists

engineers in dealing with critical issues in software development and maintenance. For example, engineers might be interested in the origins of a requirement (e.g., the stakeholder needs) or the rationale for a particular design choice. They might also need to know how exactly functional or non-functional requirements are realized in the system, or if an implementation completely realizes a given set of requirements. During system evolution and maintenance, RT is also important for analyzing the impact of new requirements or changes to existing ones. Many critical risks in software engineering are architectural [4] and have to do with persistent software attributes such as performance, reliability, or security [5]. Architectural risks have to be considered, in particular, when new requirements are assessed or existing requirements are changed. Risk assessment however is challenging and relies on understanding the complex relationship between requirements, desired system properties, and architectures during development and maintenance [6, 7], so RT becomes particularly important.

The benefits of RT are widely accepted nowadays and sophisticated tool support is available to record, manage, and retrieve trace information [8]. 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 [9].
- The full potential of RT can only be exploited if complete trace information is available. However, incomplete trace information is a reality due to complex trace acquisition and maintenance.
- It is often hard to anticipate the kind of engineering issues that might arise later and the trace information recorded for one particular purpose might be insufficient for future tasks.
- Traces have to be identified and recorded among numerous, heterogeneous engineering artifacts (document, models, code, ...). It is often very challenging to create meaningful relationships in such a complex context.
- Traces are in a constant state of flux since they may change whenever requirements or other development artifacts changes.

Automating RT can deal with many of these issues if it goes beyond mere recording and replaying of trace information [10]. We have thus been developing an automated traceability approach [11, 12] that relies on providing a small number of easy-to-find trace dependencies as input. The result of the approach are trace dependencies among various 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 significant advancement over traditional traceability techniques there is still the challenge to interpret and use the created links. We have to understand the meaning and implications of these trace dependencies (e.g., conflicts and cooperations between requirements) [13]. The manual investigation of all trace dependencies however is tedious and error prone as there

are likely thousands of links one would have to investigate in a large-scale system. We thus also propose some heuristics helping define the meaning of trace dependencies based on the types of the bridged requirements.

This work is a continuation of our earlier work on identifying trace dependencies using scenarios [11, 12, 13, 14, 15]. The contribution of this paper is on showing (1) how RT results derived by our approach can be interpreted, (2) how functional and non-functional requirements are treated, (3) how evolutionary/incremental requirements engineering is supported, and (4) how well the approach supports large-scale complex software systems.

The remainder of this paper is organized as follows: Section 2 explains our Trace Analyzer technique using the Video-On-Demand (VOD) example, a simple software package we use for the purpose of illustration. Section 3 discusses how to interpret and understand the created trace links. In Section 4, we demonstrate the capability of the approach in a case study in which we applied the technique to a large-scale open source software package. Section 5 discusses how the generated traces support various software engineering scenarios. In Section 6 we discuss related work. Conclusions and an outlook on further work round out the paper.

2. Automating Software Traceability with Trace Analyzer: The Video-On-Demand Example

Trace dependencies describe relationships between different artifacts such as requirements, designs, assumptions, rationale, system components, source code, etc. [2]. The value of recording and maintaining these dependencies is to support software understanding and to help engineers in answering questions such as "Why is this requirements here?", or "What happens if I change this design element?" Trace dependencies describe the origin, rationale, or realization of software development artifacts.

Trace dependencies are not static but highly dynamic because software evolves. For instance, if a requirement R leads to the implementation of some source code C then a trace dependency exists between the two. If the requirement changes then the source code is potentially affected. Conversely, a change to the source could make an update of the requirement necessary. This bi-directionality is very important for trace analysis and implies that if R depends on C then C depends on, at least, R.

The Trace Analyzer [11, 12] defines trace dependencies through (a) shared use of source code and (b) transitive reasoning:

(a) The source code of a software system is a useful medium to identify trace dependencies. If, say, requirement A depends on some source code C_A and requirement B depends on some source code C_B then one can infer that A and B depend on each another if C_A and C_B overlap (i.e., because they are implemented together).

(b) 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.

As input, the Trace Analyzer technique takes a small number of known or hy-

pothesized dependencies between software artifacts (e.g., requirements and source code). It then builds a graph containing nodes that capture source code and all their overlaps. For example, there are separate nodes for the source code of A and B but if they overlap then this overlap is explicitly captured in yet another node. The graph is manipulated to move known artifacts among the nodes. The goal is to constrain for all nodes what artifacts they relate to or not. 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. At the end, the graph is traversed to identify all nodes related to individual artifacts. A trace dependency is established if two different artifacts relate to at least one common node. The graph also helps in determining the strength or reliability of a dependency based on the number of nodes two artifacts have in common.

The trace analyzer technique is fully automated and tool supported. The only deficiency, as it may appear, is that some trace dependencies have to be provided as input manually; that is: traces between artifacts and code. Fortunately, we found that this input can be partially generated by executing the source code and observing the lines of code being executed. We use test scenarios to define how to test individual artifacts or groups of artifacts. When executing a scenario, we then observe which classes, methods, or lines of code are used. For instance, in a first case study we employed the commercial tool IBM Rational Pure Coverage® to observe the test scenarios of an executing system. In our 2nd case study we used a simple open source tool *org.jmonde.debug.Trace* available from http://www.geocities.com/mcphailmj/Trace/ to record the necessary trace information.

With the help of such tools, trace dependencies between test scenarios and source code can be automatically generated during testing. That is, the tool lists the methods, classes, and packages that are used during the execution of any given test scenario. If a designer now specifies how these test scenarios relate to artifacts (the premise) then one can automatically infer trace dependencies between these artifacts and the code they are using. These trace dependencies are then used as input to the trace analyzer to generate yet other trace dependencies.

2.1. *Video-On-Demand System*

We illustrate the benefits of the Trace Analyzer technique using a simple video-on-demand (VOD) system, which was developed by a third party [16]. This system provides capabilities for searching, selecting, and playing movies. It supports playing a movie concurrently while downloading its data from a remote site. VOD's complex computational logic is well-hidden underneath a simple VCR-like user interface (play, pause, stop button). Both functional and non-functional aspects are important for the requirements of the VOD.

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)*

Table 1: VOD requirements

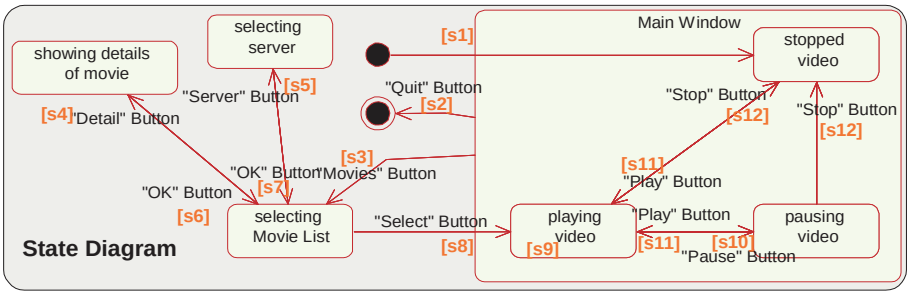

Figure 1: UML state diagram of VOD system

The VOD system consists of 21 Java application-specific classes, it also adopts numerous off-the-shelf library classes. Static and behavioral aspects of the VOD were also modeled using various UML diagrams (see the UML state diagram in Figure 1). Requirements were not available but for the purpose of extending the VOD system (discussed later) we reverse-engineered them. Table 1 depicts a subset of the VOD requirements. Figure 1 shows a state diagram of the VOD system showing that it 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 UML diagrams is that no trace dependencies are known. In some cases, trace dependencies could be guessed fairly easy but the informal nature of the requirements and the semi-formal UML model

make it hard to manually identify complete and correct trace dependencies.

2.2. *VOD Scenarios*

Our scenario-based trace analysis approach automatically defines trace dependencies among requirements, between requirements and code, and between requirements and model elements (e.g., state transitions). The approach only requires scenarios that can be tested against the code to identify trace dependencies automatically. Table 2 lists all test scenarios we defined for the case study. For example, test scenario 1 uses the VOD to display a list of movies. The details of how to test this scenario on the system are omitted for brevity but the test scenario describes how to configure the VOD system and what user interface actions to perform (e.g., which buttons to press) in order to achieve the desired results. IBM Rational's PureCoverage® was used to monitor the VOD system during the testing of the scenarios. For example, the Java classes BorderPanel (C), ListFrame (J), ServerReq (R), and VODClient (U) were executed while testing scenario 1. For the sake of brevity, we only use single letter acronyms for Java classes.

Table 2 also shows which artifacts (model element, requirements) the different test scenarios apply to. For instance, our hypothesis was that test scenario 1 relates to the state transition [s3] "Movies" Button in the state diagram (see Figure 1). While executing the scenario, it was observed to execute the Java classes (code) [C,J,R,U]. Due to transitivity of trace dependencies, one may conclude that the state transition [s3] depends on the code [C,J,R,U].

Table 2 defines 12 additional scenarios including one test scenario for each requirement (although multiple may exist) and, to make the trace analysis more realistic, some ambiguous test cases for the state diagram. We call a trace dependency ambiguous if it does not precisely define relationships among artifacts. For instance, test scenario 2 defines the state transitions [s4] and [s6] relating 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].

2.3. *VOD Trace Analysis*

Scenario-based trace analysis is fairly straightforward for unambiguous test scenarios. For instance, requirement [r6] defines a maximum delay of one second to start playing a movie. We know from test scenario 3 that [r6] executes the Java classes [A,C,D,F,G,I,J,K,N,O,R,T,U]. Consequently, this Java code needs to be optimized to perform as desired. Trace analysis becomes more complicated in case 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. Although together [s9,s11] use 13 Java classes [A,C,D,F,G,I,K,N,O, R,T,U], 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].

Test Scenario	*Artifacts*	*Observed Java Classes*
1. view movie list	[s3]	[C,J,R,U0]
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,R,T,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,U1]
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: Scenarios and observed footprints

For reasons of efficiency and precision, the trace analyzer 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. The child nodes represent subsets of parent nodes. This subset relationship applies to both the model elements and Java classes used. For instance, scenario 7 is about playing a movie [s9] and it 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.

Besides having nodes for each scenario, the footprint graph also contains nodes for all possible overlaps between scenarios. For instance, scenario 1 overlaps with scenario 5 because they both use the Java classes [C,R]. A node, child to both, is thus introduced to explicitly capture this overlap. It must be noted that some overlaps are omitted in the figure for brevity. In [11] we discuss details of building a footprint graph. For example, it is the designer's choice on how to relate the scenarios with the model elements. If the designer says that model element s9 is about scenario 7 then the designer is confident that s9 is exactly the footprint [A,C,D,F,G,I,K,O]. Thus, our trace analyzer includes s9 in the node [A,C,D,F,G,I,K,O] and it excludes every other node that this is not a subset of [A,C,D,F,G,I,K,O] (e.g., B, E, H, T, U).

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 the nodes to identify for every node the artifacts it could possibly relate to. For instance, parent node [A,C,D,F,G,I,K,O,T,U] 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 as the parent. For consistency and completeness, both children thus relate to the same model elements as their parent.

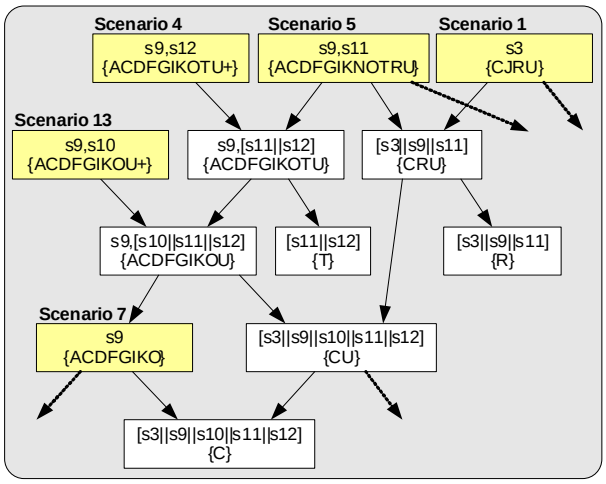

Figure 2: Partial footprint graph

In this case, the parent was defined to relate to [s9] and [s12] (ignore s3 for the moment) and thus each child individually may relate to a subset of [s9,s12]. In case of the left child, we already know that it must relate to [s9]. In fact we know that the other child [T,U] cannot be about [s9] as was discussed previously. Thus, the model element [s12] must be in at least [T,U].

The same reasoning applies to the parent node [A,C,D,F,G,I,K,N,O,T,R,U] which was defined to include [s9 and s11]. Since [s9] was defined to be exactly [A,C,D,F,G,I,K] [s9] cannot relate to [N,R,T,U] and it is excluded in that node. Given that the input required that [N,R,T,U] either belong to [s9] or [s11] and given that we now know that it cannot be [s9] we derive that node [N,R,T,U] belongs to [s11]. However, this reasoning has one flaw. While we excluded [s9] from [N,R,T,U], we did not exclude [s11] from [A,C,D,F,G,I,K]. It is quite possible that a subset of [A,C,D,F,G,I,K] has shared ownership and that subset may also belong to [s11]. This problem was addressed in [11] by explicitly tracking the possibility of shared code. This issue is also discussed in a later section on limitations of the approach.

We mentioned earlier that the approach can detect incomplete and incorrect input based on inconsistencies and incompleteness in the footprint graph. In Figure 2 we defined nodes with their included and excluded model elements. For example, node [T,U] includes [s12] but it excludes [s9] and other model elements. We put excluded elements in brackets to separate them from included elements. If a single node includes and excludes the same model element then there is a conflict which is the result of a conflicting input. Similarly, if the union of the included and excluded list does not represent the total set of model elements then the node is incomplete. For example, node [T,U] is incomplete because we do not yet know its relationship to, for example, [s7]. The trace analyzer technique uses a larger set of rules than

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	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

Table 3: Artifact to Java class dependencies

can be described in this paper and there are many special cases one has to consider to make it reliable. A detailed discussion is published in [11, 12].

All rules have in common that they move model elements within the graph structure to identify all related model elements for every node. Since in this case study the leaf nodes refer to individual Java classes, we can infer all model elements related to a Java class. Table 3 summarizes some dependencies between artifacts and code that can be interpreted from the graph. Note that leaf nodes may also refer to packages, methods, or even individual lines of code if a different level of granularity is desired by the user.

From the footprint graph we can interpret that either model element [s11] or [s12] or both have a dependency to Java class T. Table 3 shows this dependency using a letter that indicates the confidence of the trace analyzer where column T and rows [s11] and [s12] intersect: "F" for full confidence; "P" for partial confidence. The trace analyzer determined that class T either depends on [s11] or [s12]. Consequently one only has 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 input. For instance, scenario 3 in Table 2 defined [s8] to potentially depend on class F. Yet, the trace analyzer concluded that class F belongs to [s9]. Although the trace analyzer technique can reduce ambiguity, it cannot not always avoid it. Ambiguous dependencies are the result of ambiguous and/or incomplete input. The trace analyzer can also identify some forms of inconsistent input but this discussion is beyond the scope of this paper.

Trace dependencies among requirements are defined based on overlaps among the lines of code implementing those requirements. 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

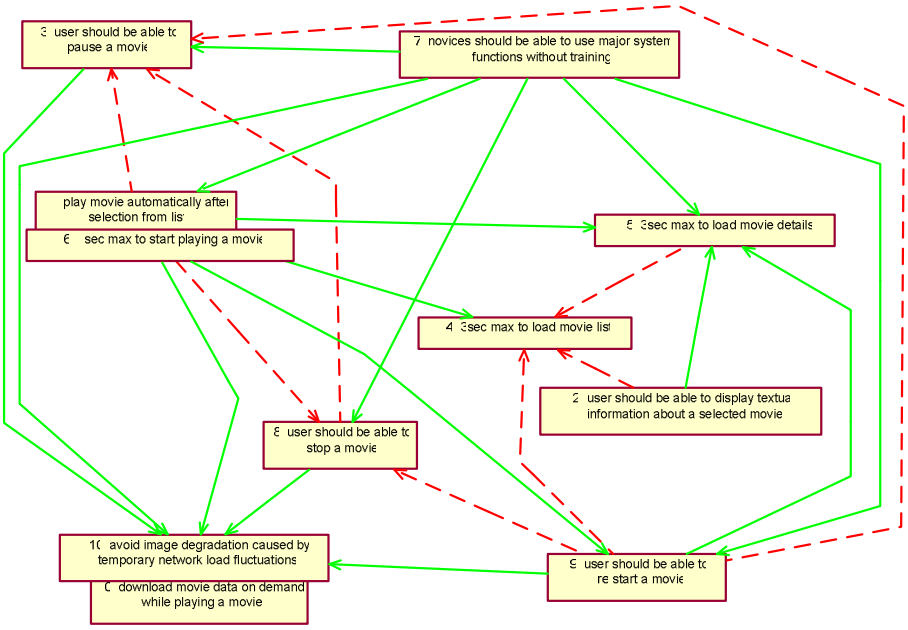

Figure 3: Automatically created trace links between VOD requirements

the footprint graph in Figure 2 to infer dependencies among requirements and/or model elements. For example, Table 2 shows a trace dependency between scenario 12 (select movie) and scenario 5 (press play button) as the latter executes a subset of the lines of code of the former. Since both scenarios represent test cases for different requirements ([r1] and [r9]), we can infer a trace dependency between [r1] and [r9].

Trace dependencies can be even inferred for quality requirements. Here, we use test scenarios executing the part of the system that is relevant for the quality requirement (i.e., the functional context of the requirements). For example, [r9] relates to [A,C,D,F,G,I,K,N,O,R,T,U] and [r6] relates to [A,C,D,F,G,I,J,K,N,O,R,T,U]. Knowing that [r9] traces to a subset of the classes that [r6] traces to we can infer a dependency between [r9] and [r6]: the requirement for a "play button" also implies the non-functional constraint of only having less than one second to start playing the movie once the button is pressed.

2.4. *Validity and Complexity of Results*

Figure 3 shows the requirements 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, a requirement pointed to by the arrow uses a subset of the lines of code of a pointing requirement. For example, there is an arrow from [r9] to [r0] because [r9] uses a subset of the lines of code

of [r0]. There are also clusters of requirements (e.g., [r1] and [r6]). Requirements in this cluster execute 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.

Here we would like to discuss two challenges:

- How can we deal with incorrect trace dependencies?
- What can we handle the high number of trace dependencies?

Incorrect trace dependencies. Figure 3 depicts some trace dependencies. Solid lines imply correct trace dependencies while dashed lines represent incorrect trace dependencies. Incorrect dependencies are identified if the code of two requirements is interleaved such that the execution of one requirement always implies the execution of the other although they to not interfere with each another; or if the granularity of the trace analysis is not detailed enough. In our case, the latter is at fault. In order to keep the presented information brief in this paper, we chose Java classes as the smallest entities. This can be problematic since different requirements may well use the same java classes although different methods thereof. In fact, if we would have done the trace analysis by comparing overlapping methods instead of classes we would not have found any erroneous results. Nonetheless, it is important to understand the meaning of trace dependencies to identify false positives. Valid results are crucial to support software understanding and trade-off analysis.

Thus far, we used our approach 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 better interconnectivity among modeling artifacts and allows 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 have to trade-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 it generally produces few incorrect trace dependencies compared to the majority of correct ones. Our approach is thus most useful in domains where completeness is desired (i.e., trade-off analysis, automation). In domains where completeness is less important than correctness, it may not suit as well.

Dealing with complexity. Figure 3 shows that trace dependencies can get very complex even in this simple example confirming the need for automation. Given complete input (i.e., if the mapping between all model elements and code is known), our approach is exhaustive in generating explicit trace dependencies among all artifact which results in a large number of (potentially incorrect) trace dependencies.

That is, if all traces between model elements and code are known then our approach generates all trace dependencies among model elements. Since there are n^2 traces among model elements for n traces to the code, it follows that we get n^2 results for n input hypotheses.

3. Some Heuristics for Understanding Trace Dependencies

As illustrated so far, the main purpose of the trace analyzer is to identify trace dependencies. These simple dependencies are already very useful for manual conflict analysis or change management. For example, we know from the example in Section 2 that requirement [r1] depends on requirement [r6] and if requirement [r6] changes then requirement [r1] is affected by this change. Even in cases of the incorrect trace dependencies we identified above (dashed lines) this reasoning is useful since the change of one requirement may unknowingly result in the change of other, dependent requirements. However, while this kind of reasoning is certainly useful, it is still superficial as it says little about how exactly requirements affect one another as pure dependencies do not convey any meaning or rationale.

3.1. *Investigating Different Types of Trace Dependencies*

The trace dependencies derived through our approach are links that merely express existing relationships but human decision makers are required to understand the true meaning of these relationships. However, upon investigating trace dependencies among requirements, we found that the meaning of these relationships is highly dependent on the types of requirements they bridge. We will discuss this finding using three examples:

Trace dependency between an efficiency requirement and a functional requirement. An efficiency requirement (time behavior) defines a time constraint on a (sub)system while a functional requirement defines user/customer requested capability. If there is a trace dependency between an efficiency requirement and a functional requirement (e.g., the dependency from [r6] to [r1]) one may infer that the execution of this particular function has to satisfy the given efficiency constraint (e.g., the movies needs to be played within one second after selection from list).

Trace dependency between two efficiency requirements. If our approach identifies a trace dependency between two efficiency requirements (e.g., the dependency from [r6] to [r5]) one may infer that the [r5] has to be at least as efficient as [r6], i.e., loading textual information about a movie has to be at least as fast as starting to play the movie. As [r5] is executed as part of executing [r6] it has to execute within the same or even better performance.

Trace dependency between two functionality requirements. If one functional requirement depends on another functional requirement an implication may be that the implementation of the second functionality requirement is a pre-requisite for the implementation of the first. In other words, eliminating the second requirement is useless if the first requirement is not eliminated either. Consider for example

Dependency Type	*Implication*
efficiency $e \rightarrow$ function f	Function f has to satisfy efficiency e
efficiency $e1 \rightarrow$ efficiency $e2$	$e2$ has to be at least as efficient as $e1$
efficiency $r \rightarrow$ security s	s needs to be realizable with efficiency r
Understandability $u \rightarrow$ Recoverability r	the recovering action r should not contradict understandability
Function $f \rightarrow$ Reliability r	f needs to be realizable within reliability r
Security $s \rightarrow$ Function f	Function f must satisfy at least security s
Security $s1 \rightarrow$ Security $s2$	$s2$ has to provide at least the level of $s1$
Function $f1 \rightarrow$ Function $f2$	Implementing function $f2$ is a pre-requisite to implementing function $f1$
Function $f \rightarrow$ Efficiency p	A part of the function f has to satisfy performance constraint p

Table 4: Implication table (examples)

requirement [r1] "play movies automatically after selection from list" and, requirement [r0] "download movie data on demand from server while playing" where [r1] depends on [r0]. Clearly, the second requirement needs to be implemented to support the first one (i.e., movie data contains information about data location, formal, and playback properties that are needed for playing).

Table 4 defines these and additional implications that are generally useful for interpreting the meaning of trace dependencies. However, one should be aware that there are exceptions. For example, two separate functionalities may be implemented "close" to one other but in a way that their execution does not affect one another. As an example, consider the requirements [r1] "play movies automatically after selection from list" and [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 towards the second one.

We need to know two things for identifying meaningful trace dependencies this way (see [13] for a detailed discussion): (1) a classification of requirements (e.g, functional, efficiency, security) and (2) trace dependencies among requirements. Given that we have automated the second part manual effort is only 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, the classification of requirements is often a byproduct of existing requirements modeling and elicitation techniques [17].

Table 5 depicts that the implication of trace dependencies can be generalized. A trace dependency between a requirement and design element is such that the requirement provides rationale for the design and the design realizes the requirement. A trace dependency between elements of a state diagram and those of a class diagram is such that the state elements provide the behavior of their classes while

Dependency Type	*Implication*
Requirement → Design	Realization of requirement
Design → Requirement	Rationale of design
Statechart → Class	Behavior of elements
Requirement → Code	Implementation
Design → Code	Implementation

Table 5: Implication table for other modeling artifacts

the classes provide the structural context of the state elements. This table is rather generic but can easily be refined.

3.2. *Analyzing the 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 4.

Figure 4 shows how the lines of code of requirements may overlap. If there is no overlap (case 1) we conclude 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 3). 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 4 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 partially (cases 2 and 3).

If a requirement uses a subset of the lines of code (case 3) of another one then the overlap is complete in one direction but not the other. 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 strong 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 3, all (correct) trace dependencies (solid links) fall in this category.

If a requirement overlaps with another requirement but it has unique source code (case 2) then the implications we can derive are weakened. For example, a security

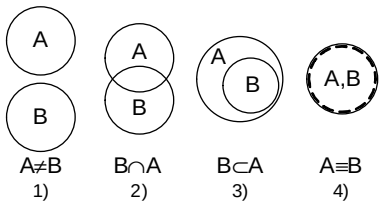

Figure 4: Types of overlaps among requirements

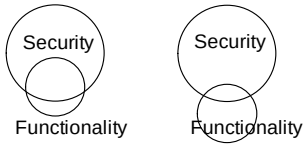

Figure 5: Partially overlapping requirements

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).

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 5) 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 such that the efficiency requirement uses most of the same Java classes as the functional one. Due to the partial overlap, we cannot imply a precise 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. *Considering Groups of Related Requirements*

An interesting extension of this discussion is to consider groups of requirements instead of individual requirements. This is not only important for the purpose of this paper but has relevance in practical settings. For example, in requirements negotiation [18] 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 but 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 6 shows that both Security1 and Security2 partially overlap with Function1.

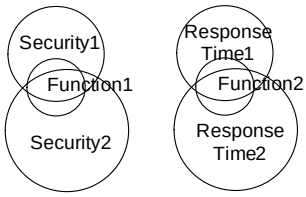

Figure 6: Grouping of requirements

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 conclude 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.

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. Case Study: ArgoUML

We performed a thorough case study for validating the trace analyzer. The software system we've chosen for our study is ArgoUML, an open source software design tool supporting the Unified Modeling Language (UML). It is written entirely in the Java programming language. The size of this software is significant containing over 1300 classes distributed in over 70 packages. The code base has over 200 KLOC. The entire source code and documentation for ArgoUML is available from http://www.argouml.org. A small subset of the ArgoUML requirements is summarized in Table 6.

The fundamental capability of ArgoUML is to support software modeling with the UML. Currently, the tool supports 8 of 9 UML diagrams. The user interface is quite intuitive and similar to other case tools for the UML. It has three major areas (see Figure 7): a diagramming area for creating and modifying graphical symbols of UML modeling elements, a model explorer for navigating in the evolving UML model, and an editor for viewing and modifying the properties of individual UML elements. ArgoUML also provides some innovative features for maintaining todo lists of open design issues, and a critiquing feature for automatically suggesting improvements to the design.

4.1. *Case Study Process*

The fundamental approach we took for the case study was to compare TA's capabilities to those of a human expert and was carried out in the following steps:

- *Selection and preparation of requirements.* The ArgoUML documentation has

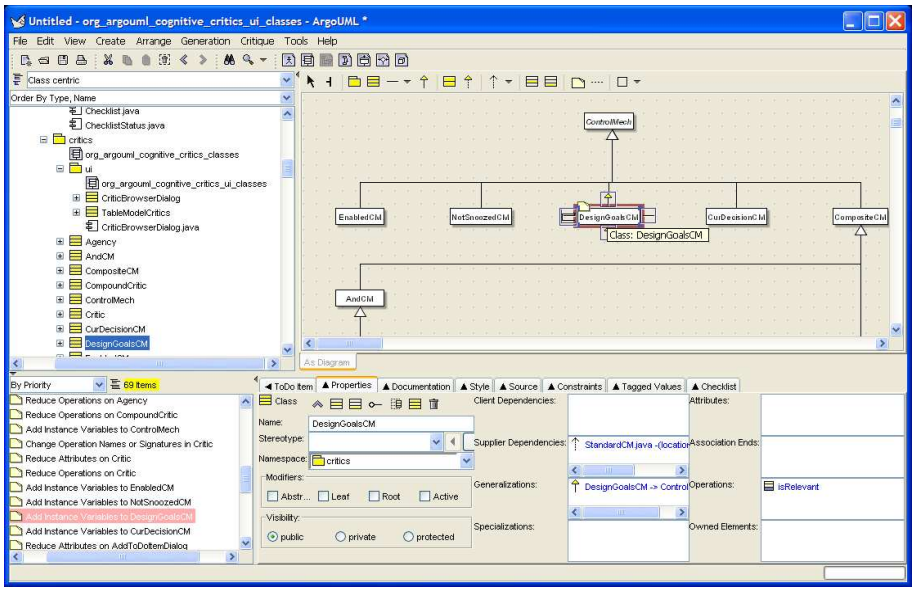

Figure 7: ArgoUML user interface

R1	The system shall allow creating a class in the current active diagram
R3	The system shall support creating a parent of the currently selected class
R5	The system shall support creating an association with a class from the currently selected class to the new class
R7	The system shall support adding an attribute to the currently selected class
R9	The system shall support creating an association between two existing classes
R14	The system shall allow to display an existing class in multiple diagrams
R17	The system shall support to use the clipboard when working with UML elements
R18	The system shall allow to find a class and navigate to it
R19	The system shall provide a zoom capability
R23	The system shall to maintain a todo-list of modelling tasks
R24	The system shall allow to load the model from a file
R26	The system shall allow to export the model to a file in XMI document format
R27	The system shall display a system information
R29	The system shall allow to create Java code for the modelled class diagram
R30	The system shall automatically critique the evolving model and provide suggestions for improving upon request
R31	The system shall allow to change the properties of a class
R33	The system shall support UML use case diagrams
R34	The system shall support UML state diagrams

Table 6: Selected ArgoUML requirements

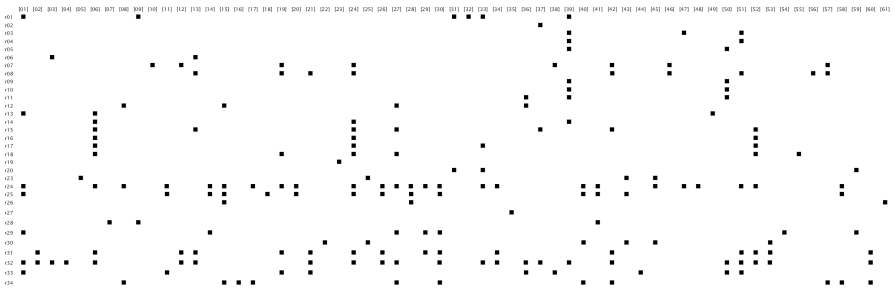

Figure 8: Requirements to source dependencies created by Trace Analyzer

some sections about the tool's requirements. Although only a few requirements are documented in the developer documentation, there are further sources for ArgoUML's requirements including a list of features, and some section in the user manuals and tutorials. In a first step we analyzed these documents and selected and compiled a list of requirements suitable for our purpose. In total 34 requirements were chosen (a selection is shown in Table 6).

- *Identification of trace links.* The next two steps were done in parallel by an engineer using TA and by a human expert identifying trace links manually. As described in the preceding sections the engineer using the TA defined scenarios and linked them to the identified requirements. He then used the tool to identify the traceability links. In parallel the human expert analyzed the requirements manually and enumerated a list of depending requirements for each of the 34 requirements.
- *Comparison and interpretation of results.* In the third step we compared the results of the tool supported trace link creation with the manual approach.

4.2. *Results*

Figure 8 shows all requirements to code dependencies identified by the trace analyzer. Row headings are the selected ArgoUML requirements (r01 to r34). Column headings are different regions of the ArgoUML code (01-61) that were found to be executed by the chosen test scenarios (those regions represent aggregations of the 1300 classes of the ArgoUML which would be impossible to visualize here individually). A total of 196 trace links were identified by the trace analyzer based on an input of only 34 trace links. Figure 8 shows the identified links between the 34 investigated ArgoUML requirements and 61 source code elements.

We compared the trace links identified by the human expert and the the tool-generated results and derived the following metrics for each requirement.

TATL: The number of trace links found by the trace analyzer.

HETL: The number of trace links found by the human expert.

	r01	r02	r03	r04	r05	r06	r07	r08	r09	r10	r11	r12	r13	r14	r15	r16	r17
TATL	16	2	12	12	9	4	12	15	9	9	10	10	12	18	16	11	13
HETL	9	6	7	7	6	3	2	2	6	6	6	2	2	2	2	1	0
OVLP	4	0	7	7	6	1	1	1	6	6	6	2	2	2	1	1	0
TAON	9	2	5	5	3	2	11	14	3	3	3	8	10	16	15	10	13
HEON	5	6	0	0	0	2	1	1	0	0	0	0	0	0	1	0	0
CONS	44%	0%	100%	100%	100%	33%	50%	50%	100%	100%	100%	100%	100%	100%	50%	100%	N/A
POWR	1,8	0,3	1,7	1,7	1,5	1,3	6,0	7,5	1,5	1,5	1,7	5,0	6,0	9,0	8,0	11,0	N/A
	r18	**r19**	**r20**	**r23**	**r24**	**r25**	**r26**	**r27**	**r28**	**r29**	**r30**	**r31**	**r32**	**r33**	**r34**		**Total**
TATL	15	0	5	3	23	20	4	0	3	12	6	18	26	17	12		**354**
HETL	2	0	0	0	2	2	2	0	0	3	0	2	6	2	3		**93**
OVLP	0	0	0	0	2	2	2	0	0	2	0	1	6	1	1		**70**
TAON	15	0	5	3	23	20	2	0	3	10	6	17	20	16	11		**283**
HEON	2	0	0	0	0	0	0	0	0	1	0	1	0	1	2		**23**
CONS	0%	N/A	N/A	N/A	100%	100%	100%	N/A	N/A	67%	N/A	50%	100%	50%	33%		**75%**
POWR	7,5	N/A	N/A	N/A	11,5	10,0	2,0	N/A	N/A	4,0	N/A	9,0	4,3	8,5	4,0		**3,8**

Table 7: Requirements to requirements dependencies

OVLP: Overlapping results, i.e. the number of requirements found by both TA and the human expert.

TAON: Number of trace links found by the trace analyzer only.

HEON: Number of trace links found by the human expert only.

These metrics allowed us to compute two further indicators for comparison.

CONS: The consistency between the human expert and TA was derived by computing what percentage of trace links found by the human expert was also found by TA.

POWR: The power of TA indicates by which factors it outperforms the human experts in terms of number of trace links identified.

Table 7 shows the identified trace dependencies among the selected ArgoUML requirements. A total of 354 links were identified by TA, the human expert came up with 93 links. 70 of these links were also identified by the TA, so the consistency of results is about 75%. The TA approach identified four times more traces than the human expert and it did so with significantly less effort. In addition, it must also be noted that using the TA approach not only produced requirements to requirements traces but it also resulted in requirements to code traces (i.e., the human expert did not produce these).

Of particular interest are the traces identified by the human expert but not the tool (HEON). These could indicate incompleteness or incorrectness on part of TA which is only possible if the test scenarios defined for the requirements were incomplete or incorrect. Or these could be erroneous trace dependencies generated by the human expert.

5. Benefits and Limitations

Automated 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 benefits of our automated approach. We also present

limitations and potential problems.

5.1. *Benefits*

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. 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.

Identifying conflicts among requirements. The following example from VOD illustrates the use of a trace dependency during requirements conflict analysis. 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 than the allowed 1 seconds max to start playing that movie. The finding of this trace dependency implies a conflict between two requirements (see Figure 9). To a casual observer, this conflict would have been hard to identify without the help of trace dependencies because it is not obvious that both requirements are related in the lines of code they execute. Once identified, a potential solution to this conflict is to change requirement [r6] to complete in less than three seconds also.

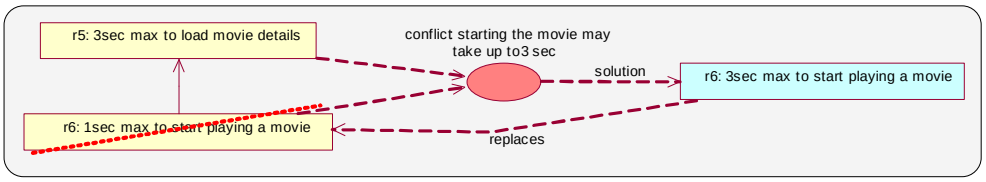

Figure 9: Trace dependency leads to a requirements conflict

r3	3 seconds max to start playing a movie
r10	Avoid image degradation caused by temporary network-load fluctuations

Table 8: New/changed requirements

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 iterative software processes [19] or in software evolution and maintenance. Normally, it is desirable to validate a new requirement or a modification of an existing requirement prior to implementation. In such a case one can still use the trace analyzer by hypothesizing about the impact of a new requirement or a requirement change. The following discusses one case of adding a new requirement that has not yet been implemented to the VOD system.

Table 8 shows the changed requirement [r6] due to the conflict with [r5] 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 temporary network congestions cause delays then this may negatively affect image quality. A possible approach to satisfy requirement [r10] would be to do some initial caching to overcome this limitation. To find out whether this new requirement clashes with other existing requirements, we can do a preliminary trace analysis. To do this, we hypothesize about the impact of the new requirement and 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 our trace analysis with this additional data.

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. As a result, [s9] fully depends on [r10]. In the following,

	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						

Table 9: Artifact to Java class dependencies (update)

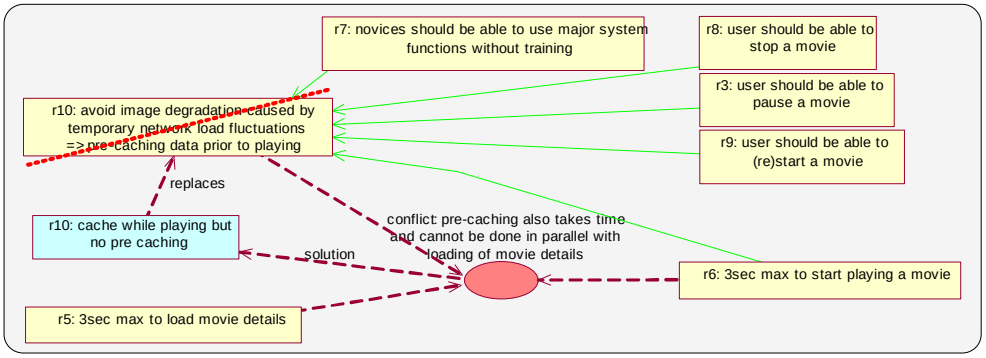

Figure 10: Trace dependency leads to a conflict with new requirement

we are more interested in the trace dependencies among the new requirement [r10] and the other requirements (e.g., [r3], [r6]) as depicted in Figure 9.

Of particular interest is the previously modified requirement [r6] which depends fully on [r10]. Recall that [r6] was changed because one second response time was considered insufficient given that at least three seconds are needed to load textual information about a movie. We thus relaxed the one second constraint to three seconds. However, now we see that if requirement [r10] gets implemented there will be additional delays. A pre-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. In this case, a potential solution is not to pre-cache movie data before playing but to incrementally build up a cache while playing. We presume that movies download faster than they are played and, consequently, it is possible to build up and increase the cache while playing. This solution might not be as effective as pre-caching but will no longer conflict with the performance requirement [r6] (i.e., note that this changed requirement would execute a different part of the system). Figure 10 visualizes the conflict and its potential solution.

Traces between requirements and design. Besides finding trace dependencies between different requirements, the trace analyzer technique also finds dependencies between requirements and design 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.

Verification of requirements. An important task of a software engineer is to determine whether the requirements have been realized properly. We specify accep-

tance 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.

Understanding level of strength of dependencies. 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. 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. i.e., 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 seconds 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.

Distinguishing domain specific code vs. generic code. The approach can also be used to distinguish project or domain-specific code from generic code. For specific analyzes 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.

Cost and Effort in Computing the Input Required. Our approach is not free but requires the engineer to define input hypotheses on how model elements (artifacts) relate to some common representation (e.g., source code). Yet, this activity is mandated in standards and it often performed by engineers to proof implementation. We also demonstrated that test scenarios may be used to ease the model element to code mapping if available. Again, engineers are required to test their systems and the overhead required to observe the test executions is small. Thus, if the source code and test scenarios are available then the cost of using our approach is solely the hypotheses on how test scenarios relate to model elements. Even this task is supported by our approach in that we allow the engineer to group model elements (e.g., [a,b] is [1,2] is easier to define then the exact value for [a] and [b] separately) or use uncertainties ([a,b] isAtLeast/isAtMost/isNode/IsExactly [1,2] as defined in [11]. Yet, in return, our approach computes trace dependencies among all model elements. Since there are over n^2 such potential dependencies, our approach computes a quadratic number of output traces for a linear number of input hypotheses.

5.2. *Limitations*

Dependent on high-quality input. The trace analyzer relies on the capability of a software engineer to relate the test scenarios to some requirements and 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 careful when doing their specifications.

Handling of shared code. A shared code is a part of the source code that is executed by two or more requirements but should not be considered an overlap during the trace analysis. For example, during the trace analysis of the ArgoUML case study, we found that a range of user interface classes existed that were triggered (executed) during testing but did not relate to the test scenario at hand (i.e., simply hovering over a user interface icon causes lines of code being executed). It is important to identify shared code to reduce the number of false positives (see also [11]). In fact, during the ArgoUML case study we've updated some TA capabilities to allow better identification of shared code. These recent extensions allow to better eliminate shared from the analysis and helps to significantly reduce the amount of "noise" generated.

Understanding of granularity. The more granular the trace analysis, the more test scenarios are needed to ensure that all parts of the source code are executed that are related (i.e., belong to a given requirement). This effort can be reduced by performing the trace analysis on less granular information (e.g., classes instead of methods). The downside of less granular trace analysis is that more overlaps exist (e.g., two requirements may use different methods of the same class and thus overlap in the common use of the same class), thus leading to more false positives. It is future work to investigate this issue in more detail.

6. 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 different artifacts (e.g., between design and code, code and documentation, requirements and architectures).

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 [20]. Basic class attributes are used as traceability anchors. The focus of this work is however not to support trace dependencies between requirements and code. Murphy *et al.* [21] aim at automating the identification of links between high-level models and source code. Their approach uses software reflexion models to find out whether an engineer's high-level model agrees with and where it differs from the source. While our approach to identifying requirements traceability is similar to design traceability [12], RT has a range of special considerations: requirements are often captured informally but they are often categorized into functional and non-functional groups (i.e., qualities). While

notations exist that express hierarchies and data/control dependencies among design elements, such notations are typically not used for requirements. A particular focus of our work is thus on the semantic implications of requirements, their hierarchies, and dependencies. Antoniol *et al.* describe an approach to automatically recovering trace information between code and documentation [22].

Gruber *et al.* discuss the problems of design rationale capture and demand the need for automatically inferring rationale information [10] from background knowledge and information captured during design. Their approach emphasizes design dependency management and rationale by demonstration. One of their key observations is related to our paper saying that rationales are not just statements of fact, but explanations about dependencies among facts.

Many approaches discuss specific traceability issues without particularly 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 [23]. 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 [24]. Pohl *et al.* describe an approach based on scenarios and meta-models to bridge requirements and architectures [7]. 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 [6].

Other traceability approaches also emphasize the automation aspect. Zisman and her colleagues have presented a rule-based approach for automatically generating and maintaining traceability relations. The artifacts and rules are described in XML and supported by a prototype tool [25]. The approach has also been applied to organizational models specified in i* and software systems models represented in UML [26].

Our trace analyzer approach generates traceability based on source code already available. A forward engineering approach complementing our approach is taken in the context of program synthesis by Richardson and Green [27]. The approach helps to automatically derive traceability relations between parts of a specification and parts of the synthesized program. The generality of the technique is demonstrated by applying it to the synthesis of Kalman Filter programs from specifications using the AUTOFILTER program synthesis system, and generation of assembly language programs from C source code using the GCC C compiler.

7. Conclusions and Further Work

In this paper we presented an approach supporting the automated generation of trace information. We discussed the approach in the context of a video-on-demand system and showed that it automates the generation of trace dependencies between the different models and artifacts of the system. We then discussed how the derived traces can support engineers in understanding software. 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 approaches. The automated approach also creates traces that 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. Another thread of our research will focus on experimenting with different levels of granularity of coverage measurement: The technique allows specifying 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 other large-scale systems.

References

1. O.C.Z. Gotel and A.C.W. Finkelstein. An analysis of the requirements traceability problem. In *1st International Conference on Requirements Engineering*, pages 94–101, 1994.
2. B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, 27(4):58–93, 2001.
3. N. Medvidovic, P. Grünbacher, A.F. Egyed, and B.W. Boehm. Bridging models across the software lifecycle. *Journal of Systems and Software*, 68(3):199–215, 2003.
4. B. W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, 1991.
5. ISO/IEC-9126. Information technology - software product evaluation - quality characteristics and guidelines for their use. Technical report, 1991.
6. P. Grünbacher, N. Medvicovic, and A.F. Egyed. Reconciling software requirements and architectures with intermediate models. *Journal on Software and System Modeling*, 2003.
7. K. Pohl, M. Brandenburg, and A. Glich. Integrating requirement and architecture information: A scenario and meta-model based approach. In *REFSQ Workshop*, 2001.
8. INCOSE. Requirements management tool survey. online at http://www.incose.org. Technical report, INCOSE.
9. B. Ramesh, C. Stubbs, and M. Edwards. Lessons learned from implementing requirements traceability. *Crosstalk – Journal of Defense Software Engineering*, 8(4):11–15, 1995.
10. T. R. Gruber and D. M. Russell. *Generative design rationale*. Lawrence Erlbaum Associates, 1994.
11. A. Egyed. Resolving uncertainties during trace analysis. In *Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), Irvine, CA, November 2004*, pages 3–12, 2004.

12. A. F. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.
13. A.F. Egyed and P. Grünbacher. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *IEEE Software*, 21(6), 2004.
14. A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Proceedings of the 17 th IEEE International Conference on Automated Software Engineering (ASE02)*, pages 163–171, Edinburgh, 2002. IEEE CS.
15. A. Egyed and P. Grünbacher. Towards understanding implications of trace dependencies among quality requirements. In G. Spanoudakis and A. Zisman, editors, *2nd International Workshop on Traceability in Emerging Forms of Software Engineering*, Montreal, 2003.
16. K. Dohyung. Java mpeg player.
17. S. Robertson and J. Robertson. *Mastering the Requirements Process*. Addison-Wesley, 1999.
18. B.W. Boehm, P. Grünbacher, and R.O. Briggs. Developing groupware for requirements negotiation: Lessons learned. *IEEE Software*, 18(3):46–55, 2001.
19. B.W Boehm, A.F. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy. Using the winwin spiral model: A case study. *IEEE Computer*, (7):33–44, 1998.
20. G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability recovery: Selecting the basic linkage properties. *Science of Computer Programming*, 40(2-3):213–234, 2001.
21. G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM, New York, 1995.
22. G. Antoniol, G. Canfora, A. De Lucia, and G. Casazza. Information retrieval models for recovering traceability links between code and documentation. In *Proceedings of the International Conference on Software Maintenance*, page 40, 2000.
23. J. Arlow, W. Emmerich, and J. Quinn. Literate modelling - capturing business knowledge with the uml,. In *UML'98: Beyond the Notation*, 1998.
24. O. Gotel and A. Finkelstein. Extended requirements traceability: Results of an industrial case study. In *Proceedings 3rd International Symposium on Requirements Engineering RE97*, pages 169–178. IEEE CS Press, 1997.
25. George Spanoudakis, Andrea Zisman, Elena Pérez-Minana, and Paul Krause. Rule-based generation of requirements traceability relations. *J. Systems and Software*, 72(2):105–127, 2004.
26. F.G. Cysneiros, A. Zisman, and G.A. Spanoudakis. Traceability approach for i* and uml models. In *Workshop Report (SELMAS 03): Software Eng. for Large-Scale Multi-Agent Systems*, 2003.
27. Julian Richardson and Jeff Green. Automating traceability for generated software artifacts . In *ASE*, pages 24–33, 2004.