Determining the Cost-Quality Trade-Off for Automated Software Traceability

Alexander Egyed
Teknowledge Corporation
4640 Admiralty Way
Marina Del Rey, CA, USA
aegyed@ieee.org

Stefan Biffl Matthias Heindl
Software Tech. & Interactive Systems
Vienna University of Technology
A-1040 Vienna, Austria
{biffl, heindl}@ifs.tuwien.ac.at

Paul Grünbacher
Sys. Engineering and Automation
Johannes Kepler University
4040 Linz, Austria
paul.gruenbacher@jku.at

ABSTRACT
Major software development standards mandate the establishment of trace links among software artifacts such as requirements, architectural elements, or source code without explicitly stating the required level of detail of these links. However, the level of detail vastly affects the cost and quality of trace link generation and important applications of trace analysis such as conflict analysis, consistency checking, or change impact analysis. In this paper, we explore these cost-quality trade-offs with three case study systems from different contexts – the open-source ArgoUML modeling tool, an industrial route-planning system, and a movie player. We report the cost-quality trade-off of automated trace generation with the Trace Analyzer approach and discuss its expected impact onto several applications that consume its trace information. In the study we explore simple techniques to predict and manipulate the cost-benefit trade-off with threshold-based filtering. We found that (a) 80% of the benefit comes from only 20% of the cost and (b) weak trace links are predominantly false trace links and can be efficiently eliminated through thresholds.

Categories and Subject Descriptors
D.2.1 [**Requirements/Specification**] and D.2.8 [**Metrics**].

General Terms
Management, Documentation, Design, Economics, Experiment.

Keywords
Experience report, Value-Based Software Engineering, Automated Trace Analysis, Cost-Quality Trade-Off.

1. INTRODUCTION
Approaches for establishing traceability between software artifacts such as user needs, requirements, architectural elements, or code play an important role in both software engineering research and practice. The topic has been researched for more than a decade [9]. Furthermore, numerous major software engineering standards such as the CMMI or ISO 15504 consider software traceability as a 'best practice' and mandate or strongly suggest the use of traceability techniques. In many branches of industry these standards are imposed on subcontractors. For example, several European car makers are demanding the fulfillment of a subset of ISO 15504 from all their subcontractors, who suddenly face the challenge of quickly and efficiently introducing traceability techniques within their organization.

Trace links support software engineers and quality assurance personnel during software development by helping them understand the many relationships among software artifacts. This is most worthwhile for large, complex, and long-living systems where there are many non-obvious relationships among artifacts. Trace analysis reveals relationships among a broader set of software artifacts such as user needs, requirements, architectural elements, or source code. For example, trace analysis can reveal which elements of a state chart or class diagram realize a requirement, or how the elements inside a state chart diagram relate to a class diagram, given that every state transition describes a distinct behavior and every class implements that behavior in form of a structure. While it might be easy to guess some of these trace dependencies, the semi-formal nature of many modeling languages (e.g., UML) and the informal nature of the requirements often make it hard to identify trace links completely.

Software traceability deals with (a) trace link generation, i.e., the manual and/or automated identification of trace links; and (b) trace link consumption, i.e., the use of trace links for conflict analysis [6], consistency checking, and change impact analysis. The benefit of these applications largely depends on the quality of the trace links provided by trace generation. For the trace analysis to be useful, one also must understand for which particular engineering tasks it is done and how incorrect trace links might affect the outcome. The ultimate goal is to tailor the trace analysis to produce the desired quality of trace links with the least amount of input effort.

Higher-quality trace links (i.e., fewer false positives/negatives) allow the users getting their tasks done faster and better. For example, higher-quality traces result in fewer false conflicts reported during requirements engineering, or a more precise definition of the impact of a change request during system evolution.

ASE'05, November 7–11, 2005, Long Beach, California, USA.

While high-quality trace links are a desirable goal, they are typically not economical to produce as identifying and validating trace links can be a big burden in a typical project context [11]. There are tools available that provide the infrastructure for managing trace links (e.g., case tools, requirements management tools). However, these tools do not free the engineers from identifying links and from ensuring their validity over time. Despite its benefits traceability is therefore hardly adopted in industry, mainly due to these cost and complexity issues [10].

To overcome these problems, researchers have been developing different automated or semi-automated approaches for trace generation [5]: For example, Antoniol *et al.* [1] 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. Spanoudakis *et al.* [13] have contributed a rule-based approach for automatically generating and maintaining traceability relations. A forward engineering approach is taken by Richardson and Green [12] in the area of program synthesis. Traceability relations are automatically derived between parts of a specification and parts of the synthesized program. These and other approaches bring some relief, but they strongly rely on the quality of the input of trace link generation: mainly, the level of detail, e.g., classes or methods as points of reference, and the level of incorrect or missing trace links.

In this paper we focus on trace generation, which deals with (1) identification of trace link candidates; and (2) validation of trace link candidates. The choice of techniques used for trace generation influences the amount and quality of traces, i.e., the number of correct trace links; incorrect trace links ("false positives"); and unreported relationships among artifacts ("false negatives"). The validation step aims at reducing the number of incorrect trace candidates. The aim is to achieve a level of quality that is sufficient for the applications consuming trace links. Of course, the desired level of quality varies with the project context and applications.

2. QUALITY CONSIDERATIONS IN TRACE ANALYSIS

Trace Analyzer [4] supports generating trace link candidates based on a small "seed set" of trace links that can come from a human expert or from logs of test cases that link requirements (and other artifacts) to source code pieces. The engineer then chooses the desired level of detail for tracing source code: packages, classes, methods, or lines of code. The Trace Analyzer simplifies the finding of trace link candidates among any kind of software artifacts (e.g., requirements, design model elements, and code). It requires as input some initial input how software artifacts relate to some common representation of the system, usually the source code. This relationship is typically explored through testing, where engineers are expected to supply a set of test scenarios that match the software artifacts. During testing engineers log the lines of code, methods, or classes that are executed by these test scenarios. Since it must be known how test scenarios match software artifacts, one can infer the software-artifact-to-code mapping.

The existence of a trace link candidate between any two software artifacts is determined by their code overlaps. If two software artifacts do not share any code part (e.g., lines of code, methods, or classes) during execution (i.e., they execute in distinct parts of the system) then no trace link is assumed among them. Note that a trace link is not a data/control dependency, but simply describes some correlation between two software artifacts. If two artifacts do share code, then there may be a trace link, if the shared code is application-specific.

Trace link candidates are thus computed by the Trace Analyzer based on the degrees of code overlap among software artifacts. Testing is a validation form that can not guarantee completeness (i.e., test cases may be missing). This naturally affects trace generation and the Trace Analyzer thus provides an input language that lets the engineer express known uncertainties [5].

The Trace Analyzer leverages manual trace generation by adding new trace candidates based on initially available trace candidates, e.g., from test suites for a requirement that relate it to the "footprint" of executed code elements. In this paper we focus on the aspect of incorrect trace candidates ("false positives") and missing trace candidates ("false negatives") as these can incur substantially higher cost for trace candidate generation and validation.

While the trace analyzer tool greatly simplifies trace candidate generation it is not fool proof and the input affects the validity and value of the output. This profoundly influences the utility of applications that consume trace links such as consistency checking, change impact analysis, or trade-off analysis during later development iterations or software maintenance. Moreover, a dilemma is that the quality of trace analysis is not just a factor of the correctness of the input. Even correct input may yield wrong traces given that we have choices in the level of detail of how trace analysis is done during development.

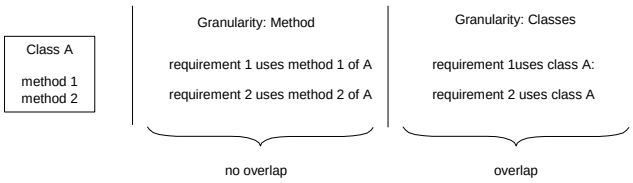

Figure 1. Code overlap example on method and class level

Take, for example, in Figure 1 two requirements that are realized by the same class but by different methods. If we analyze traces on the method-level, then the two requirements do not overlap because they are executed in distinct methods. Yet, an analysis on class level will report an overlap because both requirements relate to the same class. The statement that both requirements execute inside the same class is correct, yet, may cause incorrect traces (i.e., false positives).

Standards such as the CMMI or ISO 15504 request the use of traceability techniques and also specify useful types of trace links. However, there are no explicit statements about the required level of detail of these links. A standard might be satisfied if software artifacts are mapped to classes instead of methods even though this introduces a significant number of flaws during trace analysis. Choosing the appropriate level of detail is a difficult problem: Method-level analysis allows more detailed results than class-level analysis; lines of code allow an even finer analysis. The execution of a particular if-statement inside a method may well point to the implications of a particular requirement. From the sole perspective of trace quality, the analysis on the level of lines of code is thus even more preferable.

While traceability on finer levels of detail promises a finer picture of the relationships among software artifacts, this is achieved at higher cost. A lower level of detail typically decreases trace quality because it results in a higher number of false positives (it is hard, however, to accurately estimate to what extent). It is thus vital for trace analysis to understand the cost-quality implication of different levels of detail. How much quality do we sacrifice by saving one-order of magnitude effort in input generation (which can be translated to engineering hours)? Is the sacrifice of quality acceptable?

3. COST-QUALITY TRADE-OFF ANALYSIS IN THREE CASE STUDIES

In order to explore the cost-quality trade-offs just sketched, we conducted three case studies to investigate the effects of different levels of detail (i.e., method level, class level, and package level) for trace link generation on trace analyzer output quality (i.e., the level of false positives). The software systems we have chosen for our study are three differently-sized applications: the open-source ArgoUML modeling tool, a Siemens route-planning application, and a movie player. ArgoUML is an open-source software design tool supporting the Unified Modeling Language (UML). The entire source code and documentation for ArgoUML are available from *http://www.argouml.org*. The Siemens route-planning system, described in more detail in [10], supports efficient public transportation in rural areas with modern information technologies. The "Video on demand" package is essentially a movie player that can search for movies, select, and play them (more detailed information is available in [4]).

Table 1. Case study systems and main context parameters

System	Development context
ArgoUML	UML modelling tool; open-source development
Siemens Route Planning	Route-planning application for public transportation; industrial team work
Video on demand	Movie player application; single developer open-source development

Trade-off analysis of the level of detail vs. the level of quality. For each case study system we analyzed the impact of generating trace link candidates on the number of false positives. As baseline we took the level of false positives from trace links on the method level, the finest level of trace detail in our study. This analysis compares how a reduction of the level of detail (e.g., from method to class or from class to package) and the associated effort results in a higher level of false positives. For example, the largest of the three systems we evaluated, the ArgoUML system, has 49 packages, 645 classes, and 5,952 methods. Naturally, it is easier to map software artifacts to 645 Java classes than to 5,952 Java methods, and we can thus save at least one order of magnitude in effort by using classes instead of methods to determine the software-artifact-to-code mapping. We found that we can also save another order of magnitude, if we use packages instead of classes.

Strength analysis of trace link candidates. For the validation step after generating trace link candidates it is helpful to identify the candidates that are most likely false positives. Thus, we measured the "strength" of each trace link candidate. We expected to find a positive correlation between the number of false positives and the strength of trace link candidates.

The *"strength" of a trace link candidate* is defined as the ratio of the number of code elements (e.g., methods) that implement a requirement and the number of code elements that two requirements share as part of their implementation. For example, if requirement *X* is implemented in 10 methods, requirement *Y* is implemented in 5 methods, and 2 methods implement a part of *X* as well as a part of *Y*. Then, the trace strength for *X* is 2/10 = 20% and for *Y* 2/5 = 40%. Stronger trace links are less likely to be false positives. Our analysis once more underlines the diminishing return in investment with more detailed levels of input. Trace analysis on method level is in our study contexts around 10 times more expensive but produces almost no additional strong traces. These results are confirmed with the route-planning system data and the movie player. We found that during the validation of trace link candidates it is most worthwhile to concentrate on low-strength trace candidates. These trace candidates are harder to decide automatically and need human judgment. High-strength traces and traces with zero strength are very likely to be correct and we found in our studies little extra value in validating them manually.

4. VALIDATION SUPPORT: THRESHOLDS TO FILTER ERROR-PRONE TRACE CANDIDATES

The data analysis reported in Section 3 showed that tracing at class level can save an order of magnitude of input effort while introducing only 15-30% more false positives compared to tracing at method level. So, regarding our case studies, one could argue that the tracing quality at the class level is relatively close to tracing at method level. Yet, Section 3 showed that most of the quality problems when tracing at class level came from low-strength traces. This opens up an opportunity to investigate the option of automated filtering support (strength thresholds) for the trace candidate validation step to eliminate error-prone low-strength traces – thus improving the overall quality of the trace analysis at very little extra cost.

We experimented with three different kinds of filters, namely:

- *Threshold*: This filter eliminates all traces with strength lower than x. We applied this filter with different strength values.
- *Constant strength reduction*: This filter reduces the strength of each trace candidate by a constant.
- *Linear strength reduction*: The strength value of traces with 100% percent strength are not reduced while traces with a strength of 0% are reduced with a maximum value (e.g., 10). The strength of traces with a strength value between 0% and 100% are reduced by a linear fraction of the maximum value.

Figure 2 depicts that all filters have the most effect on weaker traces (very left side of graph), whereas strong traces (right side) are hardly affected. We found the last filter, a scaled threshold, to be most effective although the others were good also. This filter did not use a constant threshold but one relative to the traces' strengths – the weaker the trace link, the stronger the filter. The

aim of this filter was to leave strong traces strong but eliminate weak ones.

While filters are cheap, they do have a side effect. The trace analyzer does not generate false negatives, i.e., if the trace analyzer does not find a trace between two requirements, then there is none. Yet, the filters eliminate traces rather randomly with the assumption that most weak traces are false traces. Thus, a side effect of using filters is that they also eliminate some true traces. In other words, using a filter reduces the number of false positives but it introduces false negatives.

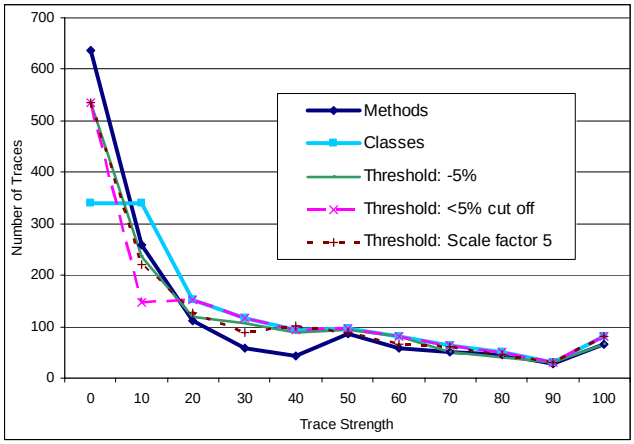

Figure 2. Eliminating false positives at class level w/thresholds

Yet, this effect has advantages because many more false positives are eliminated than false negatives are added (with small filters). That is, the weakest 10% of the trace links contained only 1% of the true traces (i.e., observe that 26% false positives may be reduced to 14% false positives by only adding 1% of false negatives). Also, some applications that consume trace links may be more amenable to false negatives than false positives. For example, consistency checking is more confused by false positives than false negatives. Thus, filtering may be used to alter the quality effect in favour of the quality needs of the consumers of trace links.

5. CONCLUSION AND FURTHER WORK

In this paper we have shown that cost-quality trade-offs play an important role when introducing traceability techniques and tools. Using three case studies we have demonstrated that it can be worthwhile to reduce the level of detail during tracing to save effort while the loss in quality might still be acceptable. Our empirical study results indicate that trace analysis on method level is roughly 10 times more expensive but produces almost no additional strong traces.

Further, we found that strong traces are very likely true traces whereas weak traces are very likely false traces. That is, the weakest 10% of all traces contain over 90% of false positives and only few true trace links. Thresholds thus can quickly eliminate false positives at the expense of also eliminating a few true trace links.

The recently defined paradigm of value-based software engineering [2][3] brings a new view into the trace analysis research area. Taking a value-based perspective can help save cost and by emphasizing investing effort on software artifacts with a perceived higher stakeholder value. Some initial results have been reported that consider value aspects in requirements traceability [7][10]. However, these approaches have not conducted a cost-quality analysis to find out when and how intensive tracing in a specific context is worthwhile.

REFERENCES

[1] Antoniol, G., Caprile, B., Potrich, A., Tonella, P., Design-Code Traceability Recovery: Selecting the Basic Linkage Properties, *Science of Computer Programming, vol. 40, no. 2-3*, pp. 213-234, July 2001.

[2] Biffl, S., Aurum, A., Boehm, B.W., Erdogmus, H., and Grünbacher, P. (eds.), *Value-Based Software Engineering*. September 2005, Springer-Verlag.

[3] Boehm, B.W.: Value-Based Software Engineering. *Software Engineering Notes, 28(2)*, (March 2003)

[4] Egyed, A. and Grünbacher, P., Automating Requirements Traceability: Beyond the Record & Replay Paradigm. P*roc. of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, Edinburgh, 2002.

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

[6] Egyed, A. and Grünbacher, P., Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help. *IEEE Software, 2004. 21(6)*.

[7] Egyed, A., Tailoring Software Traceability to Value-based Needs, In: Biffl, S., Aurum, A., Boehm, B.W., Erdogmus, H., and Grünbacher, P. (eds.), *Value-Based Software Engineering*, Sept. 2005, Springer Verlag

[8] Egyed, A. and Grünbacher, P., Supporting Software Understanding with Automated Traceability. in: *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* (in press), 2005.

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

[10] Heindl, M. and Biffl, S, A Process for Value-based Requirements Tracing – A Case Study on the Impact on Cost and Benefit, *Proc. ESEC/FSE*, Lisbon, Sept. 2005.

[11] Ramesh, B., Stubbs, L., and Edwards, M., "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.

[12] Richardson, J. and Green, J., Automating traceability for generated software artifacts. In: *Proc. 19th Int. IEEE Conf. on Automated SE*, Linz, Austria, pp. 24-33, 2004.

[13] Spanoudakis, G., Zisman, A., Perez-Minana, E., and Krause, P. Rule-based generation of requirements traceability relations. *J. Systems and Software*, 72(2):105{127, 2004.