TechRep 0607 Req Trace Strategies.doc

An Initial Tracing Activity Model to Balance Tracing Agility and Formalism

Requirements Tracing Strategies for Change Impact Analysis and Re-Testing

Matthias Heindl[1] Stefan Biffl[2]

[1] Support Center Configuration Management, Siemens Program and Systems Engineering, Siemens AG Austria, Gudrunstrasse 11, A-1100 Vienna, Austria
matthias.a.heindl@siemens.com
[2] Institute of Software Technology and Interactive Systems, Vienna University of Technology, Favoritenstrasse 9/188, A-1040 Vienna, Austria
stefan.biffl@tuwien.ac.at

Abstract. Software customers want both sufficient product quality and agile response to requirements changes. Formal software requirements tracing helps to systematically determine the impact of changes and keep track of development artifacts that need to be re-tested when requirements change. However, full tracing of all requirements on the most detailed level can be very expensive and time consuming.
In this paper we introduce an initial "tracing activity model", a framework that allows measuring the expected cost and benefit of tracing approaches. We apply a subset of the activities in the model in a study to compare 3 tracing strategies, ranging from agile "just in time" tracing to fully formal tracing, in the context of re-testing in an industry project at a large Austrian bank.
In the study a) the model was found useful to capture costs and benefits of the tracing activities to compare the different strategies; b) a combination of upfront tracing on a coarse level of detail and focused just-in-time detailed tracing can help balancing tracing agility (for use in practice) in a formal tracing framework (for research and process improvement).

Keywords: Software Requirements Tracing, Re-Test, Tracing Activity Model, Feasibility study.

1 Introduction

Main goal of software development projects is to develop software that fulfills the requirements of success-critical stakeholders, i.e., customers and users. However, in typical projects requirements tend to change throughout the project, e.g., due to revised customer needs or modifications in the target environments. These changes of requirements may introduce significant extra effort and risk, which need to be assessed realistically when change requests come up, e.g., test cases have to be adapted in order to test the implementation against the revised requirements. Thus, software test managers need to understand the likely impact of requirement changes on product quality and needs for re-testing to continuously balance agile reaction to requirements changes with systematic quality assurance activities.

An approach to support the assessment of the impact of requirements changes is formal requirements tracing, which helps to determine necessary changes in the design and implementation as well as needs for re-testing existing code more quickly and accurately. Requirements tracing is formally defined as the ability to follow the life of a requirement in a forward and backward direction [11], e.g., by explicitly capturing relationships between requirements and related artifacts. For example, a trace between a requirement and a test case indicates that the test case tests the requirement.

Such traces can be used for change impact analysis: if a requirement changes, a test engineer can efficiently follow the traces from the requirement to the related test cases and identify the correct test cases that have to be checked, adapted, and re-run to systematically re-test the software product.

However, in a real-world project full tracing of all requirements on the most detailed level can be very expensive and time consuming. Thus the costs and benefits to support the desired fast and complete change impact analysis need to be investigated with empirical data. While there are many methods and techniques on how to technically store requirements traces, there is very few systematic discussion on how to measure and compare the tracing effort and effectiveness of tracing strategies in an application scenario such as re-testing.

This paper proposes an initial *tracing activity model* (TAM), a framework to systematically describe and help determine the likely efforts and benefits, like reduced expected delay and risk, of the tracing process in the context of a usage scenario such as re-testing software. The TAM defines common elements of various requirements tracing approaches: trace specification, generation, deterioration, validation, rework, and application; and parameters influencing each activity like number of units to trace, average effort per unit to trace, and requirements volatility.

The model can support requirements and test managers in comparing requirements tracing strategies for tailoring the expected re-test effort and risk based on the parameters: process alternatives, expected test case creation effort, and expected change request severity. We apply the TAM in a feasibility study that compares effort, risk, and delay of 3 tracing strategies: value-based tracing (VBRT), no trace reuse (NTR), and systematic trace-based re-testing (FFT).

The remainder of the paper is organized as follows: Section 2 summarizes related work on requirements tracing and requirements-based testing; Section 3 introduces the tracing activity model. Section 4 outlines the industry feasibility study and Section 5 presents results. Section 6 discusses these results and limitations of the study; finally Section 7 concludes and suggests further work.

2 Requirements Tracing and Re-Testing

Several approaches have been proposed to capture traces for certain trace applications like change impact analysis, or testing [1][4][7].

2.1 Requirements Tracing Approaches

Many standards for systems development such as the US Department of Defense (DoD) standard 2167A mandate requirements traceability practice [22]. Gotel and Finkelstein [11] define requirements tracing as the ability to follow the life of a requirement in both a backward and forward direction. Requirements traceability is an issue for an organization to reach CMM level 3. Still a large number of organizations work on CMM level 1 or 2. In an assessment for reaching maturity level 3 there are questions concerning requirements tracing: whether requirements traces are applied to design and code and whether requirements traces are used in the test phases.

Watkins and Neal [23] report how requirements traceability aids project managers in: accountability, verification (testing), consistency checking of models, identification of conflicting requirements, change management and maintenance, and cost reduction.

The tracing community, e.g., at the Automated software engineering (ASE) tracing workshop TEFSE [7][8], traditionally puts somewhat more weight on technology than on process improvement. Basic techniques for requirements tracing are cross referencing schemes [9], key phrase dependencies [17], templates, RT mVBRTices, hypertext [19], and integration documents [20]. These techniques differ in the quantity and diversity of information they can trace between, in the number of interconnections between information they can control, and in the extent to which they can maintain requirements traces when faced with ongoing changes to requirements.

Commercial requirements management tools like Doors, Requisite Pro, or Serena RM provide the facility to relate (i.e. create traces between) items stored in a database. These tools also automatically indicate which artifacts are effected if a single requirement changes (suspect traces). However, the tools do not automate the generation of trace links (capturing a dependency between two artifacts as a trace), which remains a manual, expensive, and error-prone activity.

Gotel and Finkelstein [11] also state the requirements traceability problem, caused by the efforts necessary to capture and maintain traces. Thus, to optimize the cost-benefit of requirements tracing, a range of approaches focused on effort reduction for requirements tracing. In general, there are two effort reduction strategies: (1) automation, and (2) value-based software engineering.

Concerning (1) automation, multiple approaches have been developed to automate trace generation: Egyed has developed the Trace/Analyzer technique that automatically acquires trace links based on analyzing the trace log of executing key system scenarios [5][6]. He further defines the tracing parameters: precision (e.g, traces into source code at method, class, or package level), correctness (wrong vs. missing traces), and completeness. Other researchers have exploited information retrieval techniques to automatically derive similarities between source code and documentation [1], or between different high-level and low-level requirements [16]. Rule-based approaches have been developed that make use of matching patterns to identify trace links between requirements and UML object models represented in XML [24]. Neumüller and Grünbacher developed APIS [21], a data warehouse strategy for requirements tracing. Cleland-Huang et al. adopt an event-based architecture that links requirements and derived artifacts using the publish-subscribe relationship [3].

Concerning (2) value-based software engineering, the purpose of a value-based requirements tracing approach is not to reduce effort by automation but to trace all requirements with varying levels of precision, and thereby reduce the overall effort for requirements tracing [13], e.g., high-priority requirements are traced with a higher level of precision (e.g., at source code method level), while low-priority requirements are traced with a lower precision (e.g, at source code package level).

The effort used to capture traces should be justified by the effort that could be saved by using these traces in software engineering activities like change impact analysis, testing [4][15][18], or consistency checking. It is a matter of balancing agility and formalism to reach the optimal cost-benefit [2]. The approaches described above serve the purpose of reducing the effort to capture traces.

Effort for capturing tracing + effort for trace application by using traces <
effort for trace application without using traces Eqn (**1**)

Equation 1 captures this idea from a value-based perspective: to have a positive return on investment of requirements tracing the effort of generating and using traces should be lower than the effort for a trace application without traces. Such trace applications are (amongst others) change impact analysis and re-testing [10]. Besides effort also risk and delay are criteria that determine the usefulness of tracing approaches.

Changes of requirements affect test cases and other artifacts [12]. Change impact analysis is the activity where the impacts of a requirement's change on other artifacts are identified [17]. Usually, all artifacts have to be scanned for needed adoptions when a change request for a requirement occurs. A trace-based approach relates requirements with other artifacts to indicate interdependencies. These relations (traces) can be used during change impact analysis for more efficient and more correct identification of potential change locations.

In [13] we proposed an initial cost-benefit model, where the following parameters that influence the cost-benefit of RT are identified: number of requirements and artifacts to be traced, volatility of requirements, and effort for tracing. In [14] we further discussed the effects of trace correctness as parameter influencing the cost-benefit of RT. However, tracing activities were not modeled explicitly, which would facilitate a more systematic discussion of different tracing approaches.

3 An Initial Tracing Activity Model

Most work in requirements tracing research has focused more on technology than on processes supported by this technology to generate and use traces. For the systematic comparison of tracing alternatives we propose in this section a process model, the tracing activity model (TAM), which contains the activities and parameters found in research tracing approaches; the model can be used to formally evaluate and compare tracing approaches as well as their costs and benefits.

3.1 Tracing Process, Activities, and Parameters

The tracing activity model (TAM) in Figure 1 depicts a set of activities to provide and maintain the benefits of traces over time. The model is a framework to measure the cost and benefit of requirements tracing in order to compare several tracing strategies for a development project. The framework is based on previous work that identified tracing parameters, e.g., [3][7][8][13][14][16].

The activities in the model are building blocks identified from practice and literature and follow the life cycle of a set of traces: trace specification determines the scope of traces that seem relevant for a trace usage application, e.g., relationships that should be traced to support re-testing of a software application. Trace generation is the act of eliciting and organizing traces according to the trace specification; this can be a manual or (semi) automatic activity. Most approaches reported focus on improving the trace generation activity.

Trace deterioration comes from changes in the environment that may render some traces invalid and thus lower the potential benefit of the trace set. Trace validation is the activity to validate the correctness of a set of traces, i.e., to identify invalid traces for removal. Trace rework is the step to maintain a trace set, e.g., to remove or update outdated traces. Finally, trace usage is the activity, in which the traces unfold their actual benefits by saving time and effort. In the following we describe the TAM activities as basis for a model on the effort, risk, and delay of re-testing strategies and the activities’ parameters.

Trace Specification is the activity where the project manager defines the types of traces that the project team should capture and maintain. For example, the project manager can decide to capture traces between requirements, source code elements, and test cases. This activity influences tracing effort based on the following parameters: Number of artifacts to be traced, number of traces, and artifacts to be traced. Other relevant parameters are tracing scope, precision of traces [7][8][13],

Trace Generation. Trace generation is the activity of identifying and explicitly capturing traces between artifacts. Methods for trace generation range from manually capturing traces in mVBRTices or requirements management tools that automatically create traces between artifacts based. The effort to generate traces in a project depends on the following parameters:

- Number of requirements: in a software development project; the effort for tracing increases with increasing number of requirements.
- Number of artifacts to be traced to the higher the number of artifacts, the higher is the effort to create traces between them [13].
- Average trace effort per unit to trace, which depends on the used tools and the point in time of tracing.

Other relevant parameters are: number of traces, tool support, point in time of trace generation in the software development process, complexity/size of tracing objects, value of traces [13], correctness and completeness of traces.

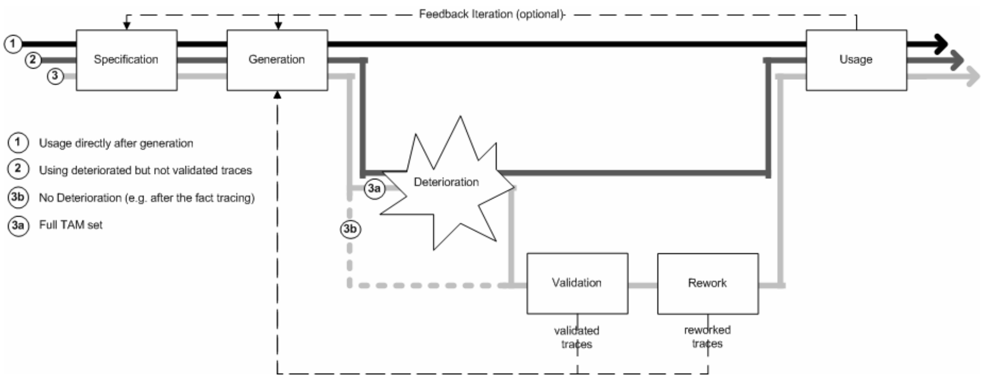
Feedback Iteration (optional)
1
2
3
Specification
Generation
Usage
Deterioration
3a
3b
Validation
Rework
validated traces
reworked traces
1 Usage directly after generation
2 Using deteriorated but not validated traces
3b No Deterioration (e.g. after the fact tracing)
3a Full TAM set

Fig. 1. Tracing activity model: Activities and process variants.

Trace Deterioration. Trace deterioration is more an event than an activity. Traces can degrade over time because related artifacts change. If only the artifacts are updated, e.g., due to change requests, and the traceability information is not updated, the set of existing traces might get invalid over time. Deterioration of traces affects the value of traces, because it reduces the correctness and completeness of traces.

Trace Validation and Rework. Trace validation is the activity that checks if the existing traceability information is valid or should be updated, e.g., identify missing trace links. In the example above, when A changes fundamentally so that there is no longer a relationship to B, a trace validator would check the trace between A and B and delete it. Trace validation is necessary to keep the trace set (traceability information) correct and up to date, so that the traces are still useful when used, e.g., for change impact analyses. We call the updating of traces trace rework. Trace validation and trace rework are often performed at once. They together make traces correct and up-to-date and counter trace deterioration.

The efforts for validation and rework depend on the volatility of requirements.

The tracing activities are not necessarily performed in sequence. Furthermore, some activities are mandatory, like trace generation, whereas other activities are optional, as indicated by the arrows in Figure 1:

- *Trace Usage directly after generation:* Trace deterioration depends on the changes made to certain artifacts. If traces stay valid over time and do not deteriorate, validation and rework are not necessary so that the existing traces can be used, e.g., for change impact analyses.
- *No Deterioration:* Traces can be validated after generation whenever the project manager wants, even when they did not deteriorate.
- *Using deteriorated traces* without validating and reworking them before is possible, but reduces the traces' benefits, because wrong or missing traces may hinder the supported activity more than they help

Trace Usage. Finally, traceability information is used as input to tracing applications like change impact analysis, testing, or consistency checking [23]. The overall effort of such a tracing application is expected be lowered by using traces. The benefits of tracing during trace usage depend on:

- Process alternatives: The value of traces during trace applications like change impact analysis or re-testing depends on the characteristics of process alternatives. For example, in our case study we compare 3 strategies for re-testing using traces; each one has different characteristics like risk, delay, or saved effort.
- Volatility of requirements: the more often requirements or other artifacts change the more likely is it that also the traces between them have to be validated and reworked. Furthermore, it may seem worthwhile to trace risky/volatile requirements in more detail, because trace applications like change impact analysis need these traces more frequently than traces to stable requirements [13].
- Expected risk: e.g., for re-testing, the risk coming from inconsistent or redundant test cases can be calculated as the expected loss of project delivery value.

The cost-benefit of requirements traceability can be determined as the balance of efforts necessary to generate, validate, and rework traces (cost); and saved efforts during trace usage, reduced risk and delay of tracing (benefits during change impact analysis). To maximize the net gain of requirements tracing the effort of generating, validating and reworking traces can be minimized, or the saved effort of trace usage can be maximized.

3.2 Research Objectives

The value of tracing comes from using the trace information in an activity such as re-testing that is likely to be considerably harder, more expensive, or to take longer without appropriate traces. If a usage scenario of tracing is well defined, trace generation can be tailored to provide appropriate traces more effectively and efficiently. Keeping traceability in the face of artifact changes takes further maintenance efforts.

The tracing activity model allows to formally define tracing strategies for a usage scenario by selecting the activities to be performed and by setting or varying the activity parameters.

In order to evaluate the feasibility of the tracing activity model we performed an initial evaluation in the finance domain, where we compared the impact of 3 re-testing strategies. Re-testing is a software engineering activity that can be supported well by requirements tracing. The goal of a trace-based testing approach can be to make testing less expensive, less risky, and to reduce the delay. For a positive return on investment of tracing the effort to generate and maintain traces plus the effort of re-testing has to be lower than the effort of testing without tracing support.

We evaluate the following research question:

- *RQ1: How useful is the TAM to model requirements tracing strategies and to estimate and compare their efforts?*

- *RQ: To what extent can we balance the agility of a re-testing approach without using traces and the formalism of a systematic tracing approach for re-testing with a value-based approach?*

To answer this research questions, we conducted a small feasibility study in a practical context, which allowed us to apply the TAM to 3 re-testing strategies, and discuss the usefulness of the re-testing strategies and the tracing model with development experts. If useful, the lessons learned from our evaluation could be a basis for extrapolation of tracing strategies and cost-benefit parameters to larger projects.

4 Application of TAM in an Industrial Feasibility Study

This section describes the study we used as proof of concept for our tracing activity model. Together with practitioners from the quality assurance department of a large Austrian bank we modeled 3 tracing strategies by using TAM building blocks and parameters and calculated tracing efforts of each strategy, their risks and delay in order to support the practitioners in deciding which tracing strategy provides the best support for re-testing in the practitioners' particular project context. This section describes how we modeled each tracing strategy just to get an overview on how we proceeded. Detailed information of the study context can be found in [TECHREP]

4.1 Empirical Study Context

The project "Default Data Base (DDB)" is a java-based, service-oriented web application for managing so-called "defaults", i.e., missing loan repayments. The project team consisted of a project leader, a technical project leader, one backend- and 2 front-end-developers, and 2 testers. Project duration was 14 months and the overall project sum around 600,000 Euro. The type and size of the project was suitable to apply the trace-based re-testing approach in a software development project with realistic yet manageable trace options. The project contained 124 functional and non-functional requirements, with candidates both for re-testing and tracing: 600 source code elements (service commands), and 700 test cases.

4.2 Tracing and Re-Testing Strategies

Table 1 describes the TAM variables that are relevant for the case study. We compared 3 re-testing strategies: no trace reuse (NTR), systematic trace-based re-testing (FFT), and ad hoc trace reuse (VBRT).

Table 1. TAM variables used in the case study

Output Variables	Description
E	Re-test Effort: Total effort of re-testing (person hours).

R	Re-test risk: Expected loss due to inconsistencies and redundancies of test cases (currency units).
D	Re-test delay: Re-testing delay (hours).
Input Variables	Description
#cr	Volatility of requirements: Number of change requests per requirement.
#tntc	Total number of test cases in the project.
#tc	Number of artifacts to be traced: Number of test cases affected by a changing requirement.
#r	Number of requirements that are traced to test cases.
NTR, FFT, VBRT	Re-testing trace process strategies.
te	Expected effort per test case to establish traces to relevant requirements (person hours).
tcn	Expected effort to develop a new test case (person hours).
tcr	Expected effort to reuse a test case (person hours).
tca	Expected effort to rework an existing test case for a changing requirement (person hours).
cia	Change impact analysis effort: Expected effort to check test cases, if they are affected by a changing requirement (person hours); tracing should lower cia.
dor	Expected effort to check existing test cases for consistency and eliminate redundancies with a new test case (person hours).

No trace reuse (NTR). As baseline strategy we used the NTR strategy, which was the standard strategy in the case study context; in this traditional re-testing strategy there is no trace support. Thus the activities of the tracing activity model are not performed and re-testing has to cope without traces: For each change request, the testers create new test cases instead of re-using and adapting existing ones. Obsolete test cases are replaced by new ones in order to avoid the risk of having redundant or inconsistent test cases, and to make sure everything is tested and test cases are still valuable after the change.

$$E(NTR) = \#cr * \#tc * tcn + dor. \qquad \text{Eqn } (\mathbf{2})$$

Equation (2) calculates the overall re-testing effort following the NTR strategy: for each change request (#cr), new test cases are created with the expected effort (#tc* tcn). Finally, the testers have to check newly created test cases with existing test cases and delete redundant (obsolete) old test cases (dor).

Systematic trace-based re-testing (FFT). In the FFT strategy, testers systematically establish traceability by relating requirements and test cases via a tool, the Mercury Test Director. When a change request occurs, they check, and adapt existing test cases whenever possible; else they create new test cases.

$$E(FFT) = \#tntc * te + ciaFFT * \#cr + tcnra * \#tc * \#cr \qquad \text{Eqn } (\mathbf{3})$$

Equation (3) calculates the overall re-testing effort following the FFT strategy: The formula consists of 3 parts: (a) upfront traceability effort (#tntc * te), which establishes traceability for each existing test case, (b) the effort to identify affected test cases for each change request (ciaFFT * #cr), and (c) the effort needed to either reuse (tcr) or adapt (tca) existing test cases, depending on the severity of the change requests (#cr). If existing test cases can neither be reused nor adapted, new test cases have to be developed (tcn).

The shares of test cases that can be reused, adapted, or need to be created anew typically has an important impact on the overall effort of re-testing.

Value-based tracing (VBRT). Strategy VBRT creates traces whenever a change request comes in, which can cause significant delay of re-testing.

$$E(VBRT) = ciaVBRT * \#cr + tcnra * \#tc * \#cr \qquad \text{Eqn } (\mathbf{4})$$

Equation (4) calculates the overall re-testing effort following the VBRT strategy: When no traces are captured, the effort to check if test cases are affected by a change request (ciaVBRT) can be expected to be significantly higher than with FFT, since no traces can be used and all test cases have to be scanned for possible change impacts. The second parts of the VBRT and FFT formulas are similar.

The option with the lowest overall effort for re-testing has the best cost-benefit. Further considerations for the benefit of a strategy are re-test risk and delay.

In the feasibility study project, we started with a simple instance of the TAM (indicated by the b-arrow in Figure 1); we focused on the activities trace specification, trace generation and the usage of generated traces for re-testing. In this initial case study the activities trace deterioration, trace validation and re-work were not enacted; rather we assumed for trace usage all generated traces to be correct. Table 2 describes the 3 tracing strategies for re-testing in the case study based on the tracing activity model.

FFT contains in comparison to the other strategies additional activities for trace specification and trace generation, which cost upfront effort, but are expected to considerably reduce the effort for change impact analysis and thereby for re-testing in general.

The following subsections briefly explain how the project team performed the tracing activities and measurement in the case study project.

Requirements documentation, source code development, and test case creation. The requirements document was a Word document containing the use case descriptions. The case study project was based on a service-oriented architecture, so the source code elements were service commands. Testers created test cases by using Mercury Test Director, which is the standard test management tool in the company. For each test case, they defined steps to perform.

Trace specification. First of all, we had to define to which artifacts we should trace the requirements in order to support testing. Finally, we decided to trace requirements to test cases and to source code elements. The traces between requirements and test cases indicate which test cases cover which requirements; the traces between requirements and source code elements indicate where each requirement is implemented.

Table 2. Characteristics of re-testing strategies NTR, FFT

Trace activity	NTR	FFT
Trace Specification	No traces captured	Detailed tracing of requirements to test cases and to source code
Trace Generation	No traces captured	M:N traces captured with Mercury Test Director
Trace Usage for Re-testing	No change impact analysis; creation of new test cases	Traces used for change impact analysis on test cases

Generally, there may exist m:n relationships between artifacts, that means one requirement may trace to multiple test cases and one test case may test multiple requirements. The same is true for source code elements and requirements.

Trace generation. The testers captured traces between requirements, source code, and test cases manually by using the test management tool Mercury Test Director. The main difference between the 3 re-test strategies is how and when dependencies between requirements and other artifacts relevant for testing are explicitly captured. Thus, all artifacts to be traced have to be represented in the Test Director.

Fortunately, the Test Director provides a requirements module, where requirements can be inserted. The requirements hierarchy consisted of 5 levels, that means each use case was refined into more low level requirements (functional and non-functional). 124 requirements were inserted in total. For each requirement the test coverage is displayed in the tool. The next step was to represent source code elements in Test Director, so that all necessary elements can be traced. As the case study project is based on a service-oriented architecture the technical components were service commands. Each service command was inserted into the tool and commented.

Trace usage (change impact analysis). In order to evaluate the trace-based testing strategy, we developed a few change requests and compared for them the traditional testing strategy (without traces) and the trace-based strategy. In the traditional strategy, NTR, the test team created a new test case for each change request that was submitted instead of adapting existing ones. In the systematic trace-based strategy, FFT, for each changing requirement, the affected test cases can be easily identified via traces and testers can adapt them in order to correctly test the changed requirement. A major question in the empirical study was whether these efforts for trace generation would be considerably lower (or higher) than the saved effort during change impact analysis and re-testing.

5 RESULTS

This section compares the 3 strategies for re-testing and their most relevant input parameters.

5.1 Comparison of 3 Tracing Strategies

Table 2 depicts the activities of the re-testing alternatives. Documentation of requirements, test case creation, and development of source code are similar in alternatives NTR and FFT, with the small exception that in the trace-based strategy, FFT, requirements and source code elements have to be inserted into the tracing tool. This is a very small one-time effort. Trace specification and generation are additional activities in the FFT strategy.

Furthermore, concerning change impact analysis, it is important how much effort can be saved by using traces in comparison to the VBRT strategy, where no traces are used.

Another important aspect for comparison is that in the traditional strategy, NTR, new test cases are created for each change request, without reusing existing test cases. In the trace-based strategy, FFT, to save effort, existing test cases are adapted and, only if necessary, new test cases are created.

Strategy NTR: Change impact analysis and re-testing without trace support

Our research question was to what extent a trace-based re-testing strategy could reduce the overall effort for re-testing.

The test process of the bank follows the NTR strategy and defines to create new test cases for each change request without reusing old test cases. In the case study context on average 6 test cases were affected by a requirements change. The total effort of NTR (see eqn. (5)) can be calculated following eqn. (2).

$$E(NTR) = 20 \text{ change requests} * 6 \text{ test cases} * 1\text{hrs} + 6*700*8 \text{ min} = 120 \text{ hrs} + 560 \text{ hrs} = 680 \text{ hrs.} \qquad \text{Eqn } (\mathbf{5})$$

In the case study context 20 change requests are typical for this type of project. The effort for creating a new test case in the project was on average 1 hour, according to test experts' estimation. Of course, this test case creation effort can vary depending on the complexity of the requirement to be tested.

The second part of the formula consists of the effort to delete obsolete test cases. Each newly created test case has to be checked against the existing ones to avoid redundancies. Checking one test case with another one took in the study on average 8 minutes. Thus, the overall effort for NTR can be calculated as 680 hrs.

Besides effort, the alternatives also differ in delay. VBRT has a larger delay, because tracing is conducted during re-test. Concerning risk, NTR would be more risky if obsolete test cases were not checked (dor); Inconsistent or redundant test case sets could then result in hidden testing effort or lower-quality test sets.

Full formal tracing FFT

The main difference between the NTR and FFT strategies is that traces are captured explicitly in strategy FFT, causing both additional upfront effort for tracing and saved effort for change impact analysis and re-testing.

A requirement can be traced to multiple test cases, and one test case can in turn be traced to multiple requirements (m:n relationships). The total effort for FFT is calculated according to eqn. (3); part 1 of the formula is the upfront trace effort, eqn. (6a); and part 2 of the formula is the effort to identify affected test cases and to reuse or adapt them, or to create additional new test cases, eqn (6b).

$$\text{Upfront trace effort} = \#tntc * te = 700 * 0{,}5 \text{ hrs} = 350 \text{ hrs.} \qquad \text{Eqn } (\mathbf{6a})$$

$$\text{CIA effort} = ciaFFT * \#cr + tcnra * \#tc * \#cr \qquad \text{Eqn } (\mathbf{6b})$$

Table 3. Re-Testing Effort Comparison

Activities	**Effort NTR (hrs)**	**Effort FFT (hrs)**
Establish Traces (Trace generation)	0	350.0
Create new test cases for every new change request and delete obsolete test cases	680.0	0
Perform change impact analysis	-	172.8
Total (hours)	**680.0**	**522.8**

The effort of FFT for change impact analysis depends on how many traces between requirements and test cases can be reused, have to be adapted, or must be created. These values depend on the type of change request, as not every change request effects artifacts in the same way, e.g., there are simple low-effort change requests, e.g., affecting locally the user interface, whereas more severe change requests may need more extensive adaptations in several software product parts.

Based on effort reports for typical change requests in the case study context we categorized change request into the classes: Mini (small), Midi (medium), and Maxi (severe). Effort profiles of these classes were:

- Mini: 70% to 90% of test cases can be reused, 10 to 30% have to be adapted, 0% made new;
- Midi: 50% to 80% of test cases can be reused, 10 to 40% adapted, 10% made new;
- Maxi: 30% to 60% of test cases can be reused, 20 to 50% adapted, 20% made new.

The efforts for reusing test cases was found to be close to 0 person hours, the effort to adapt effected a test case was on average 0.5 person hours, and the effort to create a new test case was on average 1 person hour. For a change request that means that in the worst case: for a Mini change request 30% of test cases have to be adapted, for a Midi change request 40% and 10% new test cases have to be created, and for a Maxi change request 50% have to be adapted and 20% have to be resulting in a total effort of:

Mini: 0.9 hours = (6 test cases * 30% * 0.5 hours) Eqn (**7a**)

Midi: 1.8 hours = (6 test cases * (40% * 0.5 hours + 10% * 1 hour)) Eqn (**7b**)

Maxi: 2.7 hours = (6 test cases * (50% * 0.5 hours + 20% * 1 hour)) Eqn (**7c**)

Based on empirical data from a change request sample in the case study, we found 50% of change requests to be in the class Mini, 40% to be Midi, and 10% to be Maxi. Thus, the total effort for change impact analysis with FFT was (by 20 test cases, affecting 6 test cases each):

CIA effort from Mini CRs = 50% * 6 test cases * 20 CRs * 0.9 hrs= 54 hrs Eqn (**8a**)

CIA effort from Midi CRs = 40% * 6 test cases * 20 CRs * 1.8 hrs= 86.4 hrs Eqn (**8b**)

CIA effort from Maxi CRs = 10% * 6 test cases * 20 CRs * 2.7 hrs = 32.4 hrs Eqn (**8c**)

CIAFFT effort overall = 54 + 86.4 + 32.4 hrs = 172.8 hours Eqn (**8d**)

E(FFT) = upfront trace effort + CIAFFT = 350 hours + 172.8 hours = 522.8 hrs Eqn (**9**)

Compared to the 680 person hours effort of the NTR strategy, where new test cases are created for each test case, the trace-based FFT, with person 523 hours, takes around 20% less effort. In this case the upfront investment into traceability paid off.

Value-based requirements tracing (VBRT) is a hybrid between FFT and ad-hoc tracing. Usually the upfront effort for FFT is considerably high, because all existing requirements have to be traced to test cases. VBRT tries to reduce this tracing effort by establishing traceability on a coarse level (to test case packages instead of particular test cases) and to refine them ad-hoc when necessary. That means that all requirements are traced to test case packages and when change requests occur for some requirements, the traces from these test cases are refined to particular test cases to improve change impact analysis. Equation (4) calculates the overall re-testing effort following the VBRT strategy:

E(VBRT) = upfront trace effort (on package level) + change impact analysis (VBRT) Eqn (**5a**)

E(VBRT) = 70 hrs + 325 hrs = **395 hrs** Eqn (**5b**)

The upfront tracing effort for VBRT is lower since traces have to be captured on more coarse level of detail than with FFT (70 hrs in comparison to 350 hrs with FFT). The change impact analysis effort for VBRT consists of refining traces from changing requirements to the affected test case packages. The effort for identifying particular test cases by refinement was 325 hrs. in our study resulting in a total effort of 395 hrs. for the value-based tracing strategy to support re-testing.

6 Discussion

This section discusses the limitations of the study results and lessons learned on the usefulness of the tracing activity model to compare tracing alternatives in empirical studies.

6.1 Validity of Results

The purpose of the case study was to investigate the impact of a trace-based re-testing strategy on the overall effort of re-testing in comparison to two other strategies. The strategies were modelled based on an initial tracing activity model. The activities and parameters in the model are based on related work in the tracing and value-based software engineering literature.

For practical reasons, the case study size and context was chosen to allow evaluating the approaches in a reasonable amount of time. However, the case study project setting seems typical in the company and banking sector; the project context allows reasonable insight into the feasibility of the trace-based re-testing strategy in this environment.

As with any empirical study the external validity of only one study can not be sufficient for general guidelines, but needs careful examination in a range of representative settings. Furthermore, we analysed only a simple instantiation of the tracing activity model in the case study; consisting of trace generation and trace usage, but without considering trace deterioration, and consequently neither trace validation nor rework. In practice incorrect traces and trace deterioration can considerably lower tracing benefits and need to be investigated.

6.2 Lessons Learned from the Case Study

The tracing activity model was found useful for modeling tracing alternatives, e.g., no tracing, ad-hoc tracing, systematic tracing, for certain tracing applications like re-testing. The model helps make alternative strategies comparable, as it makes the main tracing activities explicit and allows to map relevant parameters that influence tracing Some input parameters (like number of change requests in the project, or effort to create a test case had to be estimated based on practitioners' experience. Others were given by the project context, e.g., number of requirements. TAM allows to choose from the listed tracing activities and parameters and select the relevant ones to model tracing strategies for a particular usage scenario.

The calculated efforts provide a good input to reason about which tracing strategy to choose in a particular project.

In the case study context, the tracing strategy FFT was the cheapest option for supporting the trace application re-test , with similar risk as the strategies VBRT and NTR. Strategy VBRT had the largest delay, because tracing is done ad hoc during change impact analysis instead of reusing traceability information coming from development.

Of course, some of the calculated numbers have to be handled with care. For example, for the VBRT strategy we stated that each existing test case had to be checked for changes. This is theoretically true, but in practice people know most of the test cases which are likely to be affected. The challenge is that it is tacit knowledge and it is a risk if people move to another position or another employer. It will be further work to extent the TAM with this factor.

7 Conclusion and Further Work

In this paper we proposed an initial tracing activity model (TAM) as framework for defining and comparing tracing strategies for various contexts. For each tracintg activity, relevant parameters were identified from related work and mapped into the model. The model allows to systematically compare tracing strategy activities, their costs and benefits. We performed a small study in the financial service domain, where we evaluated the feasibility of the tracing activity model.

Main results of the study are: a) The model was found useful to capture costs and benefits of the tracing activities to compare the different strategies; b) for volatile projects or project parts just-in-time tracing seems favorable; c) for parts that need quick feedback detailed upfront preparation of traces can be warranted; d) a combination of upfront tracing on a coarse level of detail (e.g., package or class level) and just-in-time detailed tracing of really needed traces can help balancing tracing agility (for use in practice) in a formal tracing framework (for research and process improvement).

Further work will be to apply the TAM for studies in other contexts. We see the empirical investigation as an initial study that supports planning further empirical studies with larger projects, where other parameters of the tracing activity model, e.g., requirements volatility (number of change requests), or used tools can be varied.

References

1. G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering, 28(10):970–983, 2002.
2. Boehm, Turner. Balancing Agility and Discipline, Addison Wesley, 2005
3. J. Cleland-Huang, G. Zemont, W. Lukasik, A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability, RE 2004, 230-239
4. S. Elbaum, D. Gable, G. Rothermel, Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites, IEEE METRICS 2001
5. A. Egyed, A Scenario-Driven Approach to Trace Dependency Analysis, IEEE Transactions on Software Engineering, Vol. 29, No. 2, February 2003
6. A. Egyed, P. Grünbacher, Automating Requirements Traceability: Beyond the Record & Replay Paradigm, Proceedings 17th International Conference on Automated Software Engineering, ASE 2002, pp. 163-171,Edinburgh
7. A. Egyed, S. Biffl, M. Heindl, P. Grünbacher, Determining the cost-quality trade-off for automated software traceability, ASE 2005: 360-363
8. A. Egyed, S. Biffl, M. Heindl, P. Grünbacher, A value-based approach for understanding cost-benefit trade-offs during automated software traceability, Proc. 3rd int. workshop on Traceability in emerging forms of SE (TEFSE 05), Long Beach, California
9. M.W. Evans, "The Software Factory", John Wiley & Sons, 1989
10. P. G. Frankl , G. Rothermel, K. Sayre, An Empirical Comparison of Two Safe Regression Test Selection Techniques, Proceedings of the 2003 International Symposium on Empirical Software Engineering (ISESE'03)
11. O. C. Z. Gotel, A. C. W. Finkelstein, An analysis of the requirements traceability problem, 1st International Conference on Requirements Engineering, pp. 94-101, 1994
12. S.D.P. Harker, K.D. Eason, The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering, IEEE, 1992
13. M. Heindl, S. Biffl, A Case Study on Value-Based Requirements Tracing, ESEC/FSE 2005, 60-69
14. M. Heindl, S. Biffl, The Impact of Trace Correctness Assumptions, 5th ACM/IEEE International Symposium on Empirical Software Engineering 2006 (ISESE 2006)
15. P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, C. Chen, Behavior-based Acceptance Testing of Software Systems: A Formal Scenario Approach, IEEE, 1994
16. J. Huffman Hayes, A. Dekhtyar, S. Karthikeyan Sundaram, Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods, IEEE Trans. on Software Engineering, Vol. 32, No. 1, January 2006
17. J. Jackson, A Keyphrase Based Traceability Scheme, IEE Colloquium on Tools and Techniques for Maintaining Traceability during Design, 1991, pp.2-1-2/4
18. N. Juristo , A. M. Moreno1, S. Vegas, Reviewing 25 Years of Testing Technique Experiments, Journal Empirical Software Engineering, Issue Volume 9, Numbers 1-2 / March, 2004, Pages 7-44
19. H. Kaindl, "The Missing Link in Requirements Engineering", ACM SigSoft Soft. Eng. Notes, vol. 18, no. 2, pp. 30-39, 1993
20. M. Lefering, "An Incremental Integration Tool between Requirements Engineering and Programming in the Large", Proc. IEEE International Symp. on Requirements Engineering, San Diego, California, Jan. 4-6, pp. 82-89, 1993
21. C. Neumüller, P. Grünbacher, Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned, Proc. IEEE Automated SE 2006, 145-156
22. B. Ramesh, T.Powers, C. Stubbs, M. Edwards, Implementing Requirements Traceability: A Case Study, IEEE, 1995

23. R. Watkins, M. Neal, Why and how of Requirements Tracing, IEEE Software, vol. 11, no. 7, pp. 104-106, July 1994
24. A. Zisman, G. Spanoudakis, E. Perez-Minana, and P. Krause. Tracing software requirements artefacts. 2003.