

QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces

Antje von Knethen
Fraunhofer IESE,
D-67661 Kaiserslautern, Germany
vknethen@iese.fhg.de

Mathias Grund
AG Software Engineering,
D-67653 Kaiserslautern, Germany
grund@informatik.uni-kl.de

Abstract

Cost estimation of changes to software systems is often inaccurate and implementation of changes is time consuming, cost intensive, and error prone. One reason for these problems is that relationships between documentation entities (e.g., between different requirements) are not documented at all or only incompletely. In this paper, we describe a constructive approach to support later changes to software systems. Our approach consists of a traceability technique and a supporting tool environment. The tracing approach describes which traces should be established in which way. The proposed tool environment supports the application of the guidelines in a concrete development context. The tool environment integrates two existing tools: a requirements management tool (i.e., RequisitePro) and a CASE tool (i.e., Rhapsody). Our approach allows traces to be established, analyzed, and maintained effectively and efficiently.

1. Introduction

Software systems have to be changed under various circumstances during development and after delivery. To estimate costs of change requests and to implement change requests consistently, impact analyses have to be performed.

Impact analyses provide entities that are potentially affected in a system by proposed changes [1]. In our understanding, a system consists of code and documents at higher abstraction levels, for example, system requirements and system design.

There are different approaches to analyze the impact of change requests. Traditionally, impact analysis has been something that software professionals do intuitively, after some cursory examination of the code and documentation. This may be sufficient for small systems, but it is not so for large and critical ones. In addition, empirical investigation shows that even experienced software professionals predict incomplete sets of change impacts [2]. Another approach to come up with a set of change impacts is to analyze relationships between and within development documents. Arnold and Bohner [1] identified two major technology areas to identify relationships between and

within development documents: dependency and traceability analysis. Dependency analysis involves examining detailed dependency relationships between program entities. Traceability analysis involves examining dependency relationships between all types of documentation entities. In general, dependency analysis provides a detailed evaluation of low-level dependencies in code, whereas traceability analysis provides a coarse evaluation of relationships between all types of documentation entities.

In this paper, we suggest a fine-grained traceability technique and a supporting tool environment. In contrast to other traceability techniques, our technique gives detailed guidance on what traces should be established to support later impact analyses. Furthermore, the technique defines who should establish traces when as well as who should analyze traces how. The tool environment integrates a requirements management tool and a CASE tool. This integration allows the approach to be applied in a concrete project environment. In general, our approach allows traces to be established, analyzed, and maintained effectively and efficiently. So far, our approach provides support for the requirements engineer, project planner, and maintainer.

This paper is structured as follows: Chapter 1 describes the different roles that are involved in the change process and their problems: the requirements engineer establishes traces, the project planner estimates the costs of change requests, and the maintainer implements change requests. Chapter 2 introduces our trace model and describes what traces should be established and how this could be done efficiently. Chapter 3 introduces the tool environment and explains its functionality. Chapter 4 describes how the activities of the different roles are improved with the help of our tracing technique and the tool environment. Chapter 5 discusses related works. Chapter 6 gives a conclusion and describes future research.

2. Roles and activities

Changing software systems involves different development and management roles.

In this paper, we focus on three roles that are supported by our approach: (1) requirements engineer, (2) project planner, and (3) maintainer. Other roles that are involved

in changing, such as tester or designer, are not supported yet and, therefore, are not introduced in the following.

Figure 1 gives an overview of the three roles investigated and their main activities concerning changing software systems. The requirements engineer elicits, documents, and maintains requirements in requirements documents. The project planner has to understand proposed changes and analyze traces within the software system to estimate the costs of proposed changes. The maintainer has to analyze traces within the software system to implement accepted changes consistently. In the following, we assume that certain commercial tools are available to support the roles: a requirements management tool (e.g., *RequisitePro* [3]) for documenting, managing, modifying, and analyzing natural language requirements and a CASE-tool (e.g., *Rhapsody* [4]) for documenting, managing, modifying, and analyzing object-oriented system models (e.g., use cases, class diagrams, and statecharts).

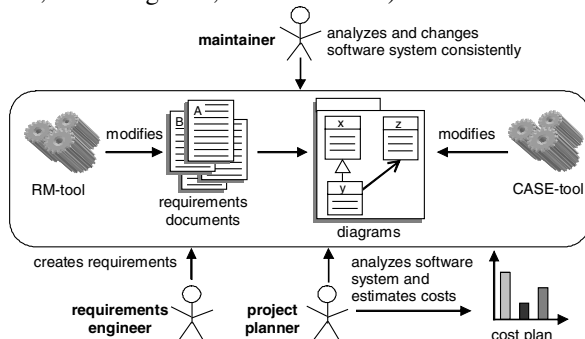


Figure 1. Roles supported by our approach

In the next sections, we describe each of the three investigated roles in more detail and give examples for their typical working steps. The detailed description and the working steps result in questions that have to be tackled by each role. The following chapters describe how our approach answers these questions.

2.1. Requirements engineer

The requirements engineer elicits, documents, and maintains requirements. Requirements are described in different notations: there are, for example, structured natural language requirements and/or UML diagrams, such as use cases, class diagrams, etc. The requirements engineer integrates different types of the requirements into a set of requirements documents (at different abstraction levels and from different views), keeps different requirements consistent, and establishes traces between them. Figure 2 shows the working steps of a requirements engineer when integrating new requirements.

In the following paragraph, a concrete working scenario of the requirements engineer is given.

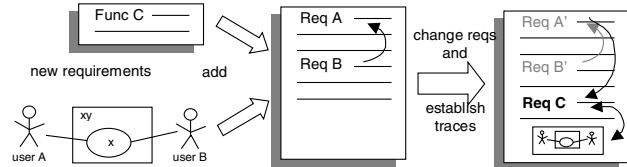


Figure 2. Working steps of the req. engineer

The requirements engineer created the “system requirements document” with support of a requirements management tool (e.g., *RequisitePro*). Now, s/he wants to document the new functionality *Func C*, in form of a natural language paragraph and a use case included in a use case diagram. First, s/he describes the functionality textually with a word processor, for example, *Word*. Then, s/he integrates the paragraph into the requirements management tool to ease the management of the requirement. After this, s/he models a use case as a part of a use case diagram with the help of a CASE tool (e.g., *Rhapsody*). In a further step, s/he changes existing requirements affected by the new functionality. Then, s/he establishes traces between related requirements (e.g., a trace between the natural language paragraph and the use case that describes both the functionality *Func C*).

During the working scenario, the requirements engineer tackles with following questions:

- How can non-textual requirements be integrated into a requirements management tool?
- What granularity of entities should be traced?
- What traces should be established?
- How should traces be documented?
- How can the effort for establishing traces be reduced?
- How can traces be established between documentation entities managed by different tools (e.g., natural language paragraph and use case)?

2.2. Project planner

The project planner estimates the effort and costs of proposed changes. Based on the cost and effort estimation, it is decided whether the change will be performed or not. To estimate the effort and costs, s/he analyzes the impact of the proposed changes on the software system. S/he performs an impact analysis by analyzing established traces within the software system. Such an analysis results in a set of documentation entities (so-called change impacts) that are potentially affected by a proposed change. The change impacts will be rated concerning the effort and costs of changing. Figure 3 shows the steps, when estimating the costs of proposed changes.

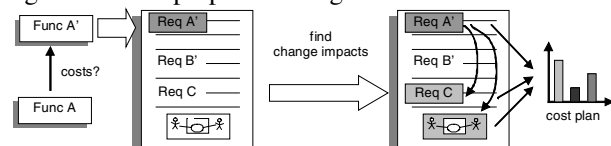


Figure 3. Working steps of the project planner

In the following paragraph, a concrete working scenario of the project planner is given.

The functionality *Func A* has to be changed. The project planner analyzes the corresponding requirement *Req A'* in the requirements document. For this, s/he uses the analysis facilities of the requirements management tool. S/he identifies two further requirements, which might be affected of the changes: *Req C* and a use case of a use case diagram. Both have a trace to *Req A'*. S/he rates the effort and costs for changing *Req A'*, *Req C* and the use case, and so, finally, estimates the costs of the proposed change.

The analysis algorithm chosen by the project planner in the example analyzes dependency relationships of the starting impact set derived from the proposed change to directly related entities (but dependency relationships of related entities are not examined further). In addition, representation and refinement relationships of all impacts predicted are investigated.

During this working scenario, the project planner tackles with following questions:

- Are all traces established?
- Do established traces result in all actual change impacts?
- How should established traces be analyzed to determine a set of change impacts with as less effort as possible?
- How can the effort and the costs of changing change impacts be estimated?

2.3. Maintainer

The maintainer integrates accepted changes into the existing software system. To do so, s/he checks all requirements marked as change impacts by the project planner, whether and how they have to be changed exactly. For each actual change impact, the maintainer analyzes all documentation entities related and checks whether they are affected, too. At least, s/he changes the change impacts consistently. Figure 4 shows the working steps of the maintainer, when implementing changes.

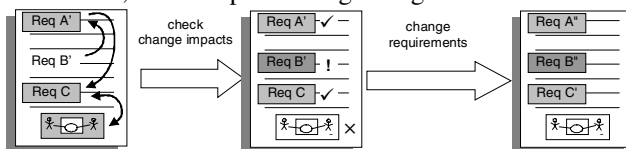


Figure 4. Working steps of the maintainer

In the following paragraph, a concrete working scenario of the maintainer is given.

The maintainer integrates an accepted change to *Func A*. The impact analysis of the project planner resulted in the change impacts *Req A'*, *Req C* and the use case. The maintainer checks whether these impacts are really affected by the change and identifies that only *Req A'* and *Req C* have to be changed. When changing the actual change impacts with the help of the requirements management tool, s/he analyzes and identifies a trace between *Req B* and *Req A*. Because of the modified interface of *Req*

A', *Req B* has to be changed, too. A further analysis of related requirements of *Req B*, has no results.

The maintainer has to tackle with following questions:

- Are all traces established?
- Do established traces result in all actual change impacts?
- How should established traces be analyzed to determine a set of change impacts with as less effort as possible?
- How could established traces be maintained with as less effort as possible?

3. Tracing approach

A tracing approach captures and manages relationships between documentation entities. A tracing approach is used to establish traceability. Traceability [30] is the ability to determine which documentation entities of a system are related to which other documentation entities according to specific relationships. A tracing approach defines which entities, which granularity, and which relationships should be traced by whom and when.

There are different tracing approaches that support different roles, for example, the tester [14], the designer [15], or the requirements engineer ([16], [18]).

3.1. Trace model

A conceptual trace model, also called reference model [5], defines what trace entities and traces are and which traces should be captured. Concerning the entities that should be related, (1) the kind, (2) the granularity, and (3) attributes that should be added have to be defined. The kind describes which documents (e.g., requirements, test cases, or design) should be involved in the trace model. The granularity determines the granularity of the entities involved (e.g., paragraphs or sentences of a textual requirements document). Each documentation entity can be extended by attributes, such as source of the entity or status (e.g., incomplete, complete, or verified). Concerning the relationships to be documented, (1) the kind and (2) attributes that should be added. The kind of relationship describes which relationships are involved in the trace model (e.g., dependency, refinement, or representation relationships). Relationships can be extended by attributes, such as status, completion date, or authorization.

Figure 5 gives a subset of a typical conceptual trace model defined by a tracing approach [5].

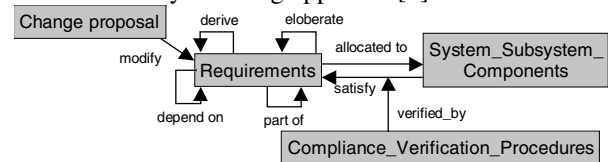


Figure 5. Subset of a trace model

This model does not give much guidance for establishing traces. It distinguishes only between different high-level types of documentation entities (e.g., requirements,

non-functional requirements, system components). Therefore, the model cannot prescribe multiplicities that describe how many entities have to be related. It only describes, for example, that a requirement has to be allocated to system subsystem components but it gives no guidance on what types of components and how many components should be related to a certain requirement. Therefore, such as a model does not guarantee that traces are established objectively and that all traces are documented.

In contrast to existing trace models (e.g., [5]) our model uses domain-know-how to determine low-level types of documentation entities (use case, monitored variable, system function). In addition, we distinguish between three general types of relationships between documentation entities: representation, refinement, and dependency relationships.

- *Representation relationships* are relationships between two documentation entities that represent the same logical entity (e.g., a natural language paragraph describing *Func C* has a representation relationship to a use case describing *Func C*).
- *Refinement relationships* are relationships between documentation entities that represent different logical entities at different levels of abstraction (e.g., a use case describing *Func C* has a refinement relationship to a use case describing *Func X* that refines *Func C*).
- *Dependency relationships* are relationships between documentation entities that represent different logical entities at the same level of abstraction (e.g., a use case describing *Func C* has a dependency relationship to an actor *Actor A*).

In the literature, four types of general relationships can be found: (1) Representation relationships are used, for instance, in [16] for finding inconsistencies between different views (2) refinement relationships are used, for example, in the four variable model [21], [22] and for decomposing requirements [23]. An example for (3) dependency relationships can be found in [19]. (4) Evolutionary relationships, relationships between entities of different versions of a documentation entity, are used for example in REMAP [25]. The approaches found in literature typically use several types of relationships. For instance, the RE framework [20] uses all and [31] uses representation, refinement and dependency relationships.

Figure 6 shows a subset of our conceptual documentation model. It describes a representation relationship between a paragraph “*function overview*” in the chapter “*overview*” and a “*use case*” of a “*use case diagram*” of the chapter “*functional description*”.

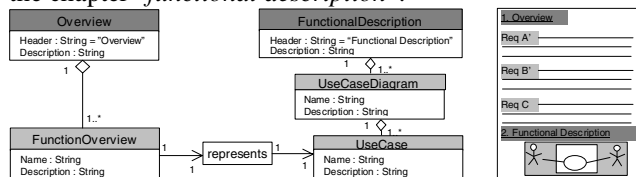


Figure 6. Subset of our concept. documentation model

These low-level types of documentation entities, the knowledge about the logical entities described by the documentation entities and the distinction between the general types of relationships allow explicit multiplicities to be defined in the conceptual documentation model. Therefore, more guidance for requirements engineers can be provided. Our model supports objective setting of traces, eases that all relationships are documented and that all established traces are right. More details on how we defined our trace model are described in [6], [7], [10].

3.2. Relationships guidelines

From the conceptual documentation model, we derive guidelines that describe what traces should be established. The guidelines support establishing traces by the requirements engineer. Figure 6 describes, for example, a representation relationship between a paragraph in the chapter “*overview*” and a “*use case*” in the chapter “*functional description*”. Based on this information, two guidelines are derived. We structure our relationship guidelines according to the structure of the requirements document:

- Chapter “*overview*”: each overview description of a function described in this chapter requires a representation relationship to a use case.
- Chapter “*functional description*”, section “*use case diagram*”, “*use case*”: each use case described requires a representation relationship to a paragraph “*function overview*” included in chapter “*overview*”.

Such guidelines are derived for each relationship defined in the conceptual documentation model.

3.3. Tracing guidelines

The tracing guidelines describe how traces determined by the relationship guidelines should be documented. Therefore, the tracing guidelines give additional information to each relationship guideline.

We distinguish three stability degrees of documentation entities: (1) completely or partly predefined, (2) predefined at a type level, and (3) completely variable.

- A documentation entity can be completely predefined, if the information appears in all requirements documents of a type in a certain domain. For example, the headers of the different chapters of a requirements document shall always be the same. Completely predefined documentation entities can be formulated as part of the template or can be generated automatically.
- A documentation entity can be predefined at a type level, if the same type of information appears in all requirements documents of a type in a certain domain. For example, the chapter “*overview*” includes always a set of functions (type). The concrete functions (e.g., “*seat control*” (instance)) included depend on the system described by the requirements document. Based on the knowledge that a documentation entity is predefined at a

type level, constraints can be described that guide the requirements engineer through filling out the requirements document template. The constraints describe what information should be described by the documentation entity.

- A documentation entity is completely variable, if the entity describes different information in various requirements documents of a type in a certain domain. For example, the documentation entity “*description*” of the documentation entity “*users of system*” includes different information from document to document. For this type of documentation entity, no support can be given.

We suggest establishing traces between documentation entities of different degree of stability as follows:

- Predefined documentation entities and relationships between them will be provided as part of a predefined template.
- For predefined documentation entities and relationships at a type level, a set of guidelines are integrated into the template (see relationship guidelines).
- For completely variable documentation entities, no guidance is given.

Relationships between documentation entities can be documented explicitly but also implicitly. Our tracing guidelines reduce the necessity for explicit link setting. This reduces the effort for establishing traces. We use *implicit links* as described in [8], [9] namely name tracing, and relationships given by documentation structure. In particular, all representation relationships can be established through name tracing such that all documentation entities related to the same logical entity could be identified by the same name. All refinement relationships can be established through documentation hierarchy. That means that the documentation entity corresponding to the logical child entity is part of the documentation entity corresponding to the logical parent entity.

4. Tool environment “QuaTrace”

Our tool environment *QuaTrace* [12] allows the efficient application of the relationship and tracing guidelines in a concrete development context. *QuaTrace* supports the requirements engineer, project planner, and maintainer in their activities by offering functionality helping to automate parts of their work, such as establishing and analyzing traces. *QuaTrace* integrates and extends two commercial tools: a requirements management tool (i.e., *RequisitePro*) and a CASE tool (i.e., *Rhapsody*). We chose *Rhapsody* as a CASE tool because of its simulation facilities. *RequisitePro* was taken because of its high integration in *Word*, which allows requirements documents and requirements to be described directly in *Word*. We also had a look at the requirements management tool *DOORS* [11], which is much more powerful than *RequisitePro*, especially in linking requirements, but offers less comfortable

editing of requirements as *RequisitePro*. While *RequisitePro* and *Rhapsody* offer both a COM-API, which allows accessing both tools within one program, *DOORS* offers an own script language.

A list of commercial traceability tools can be found in [26], e.g. Caliber RM, CORE, DOORS and RequisitePro. Examples for research tools are TOOR [28], PRO-ART [17], STAR-Track [27]. TOOR, uses relations instead of simple links. It allows the user to define meaningful relations for the kind of connection being made. In addition, the tool allows multiple and flexible ways to trace requirements including both browsing and regular-expression search. PRO-ART allows different relationships of the defined three dimensions of requirements engineering to be managed. By using internet tools STAR-Track reduces the costs for commercial tools. In addition, it supports heterogeneous computing environment.

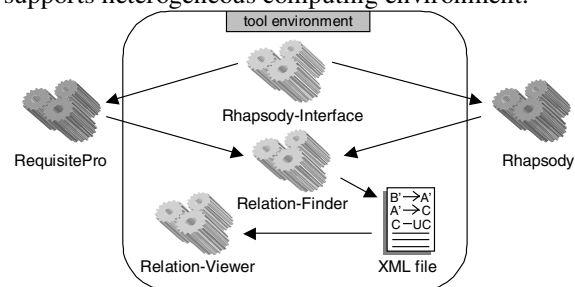


Figure 7. Components of the tool environment

Figure 7 shows the components of *QuaTrace*. The component *Rhapsody-Interface* reads *Rhapsody*-projects to export UML-elements to *RequisitePro*. *Relation-Finder* analyzes *RequisitePro*- and *Rhapsody*-projects to extract traces established with *RequisitePro* or *Rhapsody*. Traces are stored in a XML file. *Relation-Viewer* visualizes the stored traces and allows the traces to be analyzed from different viewpoints.

The components of *QuaTrace* offer the following main functionalities:

1. Interface between *RequisitePro* and *Rhapsody*

The requirements engineer has to establish traces between natural language requirements and UML elements (e.g., use cases). For this, s/he needs an interface between *RequisitePro* and *Rhapsody*. There was no interface between both tools. Therefore, we have to develop the tool *Rhapsody-Interface*. For each UML element that is managed by *Rhapsody* and that has to be traced, *Rhapsody-Interface* tool creates a so-called shadow-requirement in *RequisitePro*. By representing *Rhapsody*-elements as shadow-requirements, it is now possible to establish traces from requirements in *RequisitePro* to UML elements in *Rhapsody*.

2. (Semi-)automated setting of traces

Establishing traces is guided by our relationship and tracing guidelines but is still time consuming for the requirements engineer. The developed tool *Relation-Finder* analyzes *RequisitePro*-projects to identify estab-

lished traces. The advantage is that the requirements engineer no longer needs to set traces manually. By using name tracing the requirements engineer only needs to set implicit links, *Relation-Finder* identifies the traces between different requirements and sets them explicitly. For being independent of the tracing capabilities of the used requirements management tool, the established traces are stored in an XML file.

3. Tailored (impact) analysis

A manually performed impact analysis is error prone and time consuming. The maintainer needs information about all related documentation entities of a selected requirement. The developed tool *Relation-Viewer* analyzes all traces and orders change impacts according to the type of relationship. Distinguishing three types of change impacts supports the cost estimation of the project planner.

In the following sections, we describe the developed components in more detail.

4.1. Rhapsody-Interface

The tool *Rhapsody-Interface* implements an interface between *Rhapsody* and *RequisitePro*. The tool exports *Rhapsody* elements to *RequisitePro* as so-called shadow-requirements, placeholders in *RequisitePro*, which represent an UML element in *Rhapsody*. For every exported UML element, a shadow-requirement in *RequisitePro* is created. The name of the created requirement consists of the element type of the UML element, e.g. *class* or *attribute*, and its name. The shadow-requirements reflect the hierarchy of the UML elements. An attribute, for example, is a sub-requirement of a class, a class a sub-requirement of a package.

Figure 8 shows an example for UML elements and its exported shadow-requirements. The package *Demonstration* and its elements are exported to the shadow-requirements with the tag *Req1*, *Req1.1* ... The name of the shadow-requirement of use case diagram *Demonstration* is according to its metaclass *UseCaseDiagram* *Demonstration*.

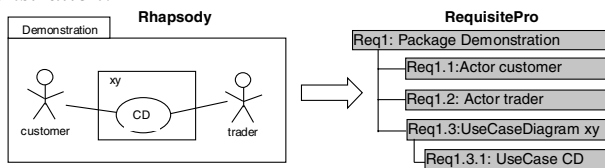


Figure 8. Example for exported UML elements

Those placeholders allow UML elements to be used as 'normal' requirements in *RequisitePro* and establishing and analyzing traces. For each exported element, export information is stored in the *Rhapsody* element and in the *RequisitePro* shadow-requirement, too. By this, a consistency check is possible, detecting accidentally deleted shadow-requirements, renamed or deleted exported UML elements in *Rhapsody*. Furthermore *Rhapsody-Interface*

can export diagrams as wmf-file. The selection of *Rhapsody* elements can be done manually, by selecting each element of a *Rhapsody* project, or by metaclass. That means all *Rhapsody* elements of a specific type, such as *class*, *attribute* or *diagram*, are automatically selected for export.

Figure 9 shows the user-interface of the *Rhapsody-Interface*. On the top, one can choose, create or delete presets. Presets are predefined settings for the *Rhapsody-Interface*, for example, which UML elements should be exported. In the middle and on the right side, one sees the consistency options and the options for selecting the *Rhapsody* elements for export.

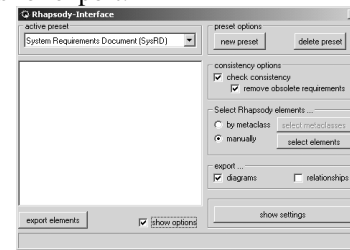


Figure 9. Rhapsody-Interface

Rhapsody-Interface supports the requirements engineer by helping her/him to establish traces between natural language requirements and *Rhapsody* elements.

4.2. Relation-Finder

The tool *Relation-Finder* analyzes the requirements of a *RequisitePro*-project for implicitly and explicitly established traces and stores them in an XML-file.

Relation-Finder offers several options to analyze requirements:

1. Analysis of *RequisitePro*-traces

RequisitePro allows the user to establish two types of traces: refinement and dependency traces. Because of *RequisitePro* does not allow circular relationships, the setting of relationships in *RequisitePro* is possible, but in our case too restrictive. *Relation-Finder* reads all established traces and stores them as refinement, respectively dependency traces.

2. Analysis of *Rhapsody*-traces

Each shadow-requirement in *RequisitePro* is a representative of an UML element in *Rhapsody*. The relationships between the UML elements are stored as refinement, if they are generalization or aggregation, and as dependency, if they are association or dependency relationships.

3. Analysis of implicit links

Using name tracing sets implicit links. Requirements describing the same logical entity have the same unique name. When referencing to a logical entity in a textual description, always the unique name of the logical entity is used. If two requirements use the same name, *Relation-Finder* stores an explicit link between both requirements. If the name or the tag of a requirement is

used in the text of another requirement, then this is interpreted as a dependency trace.

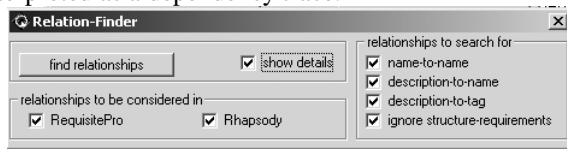


Figure 10. Relation-Finder

Figure 10 shows the user-interface of the *Relation-Finder*. On the right side, there are the options for the analysis of implicit links. At the bottom, one has the options, which tools to be considered in analysis.

All requirements and found traces are stored in a XML file. Storing traces in an external file was necessary, because of the limited linking capabilities of *RequisitePro* that does not allow circular traces and attributes of traces to be defined.

Relation-Finder supports the requirement engineer by establishing all traces s/he set implicitly.

4.3. Relation-Viewer

The tool *Relation-Viewer* visualizes the requirements and their relationships in a tree structure. *Relation-Viewer* has two modes: *maintainer-view* and *project-planner view*.

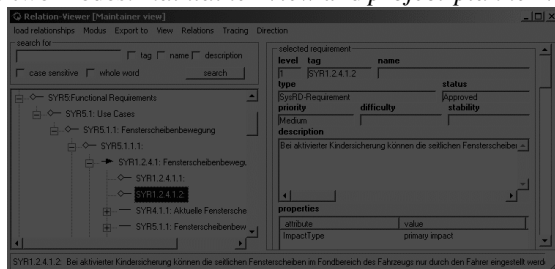


Figure 11. Maintainer-View

The *maintainer-view* shows all requirements and traces. By selecting one requirement, all traces of this requirement are shown. *Relation-Viewer* provides the possibility to search for requirements and to filter traces. You can filter the type of trace – representation, dependency or refinement –, the direction – directed or symmetric/bi-directional –, and in- or outgoing traces.

Figure 11 shows the *Maintainer-View* of *Relation-Viewer*. On the left side, there is a tree view of the requirements and their traces; on the right side, you see details on a selected requirement and traces.

This *project planner-view* offers a special impact analysis. If the project planner has selected a certain requirement that have to be changed (i.e., starting impact set), *Relation-Viewer* analyzes the traces to identify change impacts. Identified change impacts are structured according to the type of trace and separated for each requirement document. The separated types are:

1. Primary impacts

Primary impacts are all documentation entities that have to be changed to implement a proposed change consist-

tently: The set of primary impacts consists of the starting impact set and all documentation entities related by representation relationships.

2. Secondary impacts

Secondary impacts are all requirements that have to be changed with a high probability. The set of secondary impacts consists of all documentation entities that are related by refinement relationships to the primary impacts and their representations.

3. Tertiary impacts

Tertiary impacts are all documentation entities that might be changed. The set of tertiary impacts consists of all documentation entities that are related by first-order dependency relationships to the primary impacts.

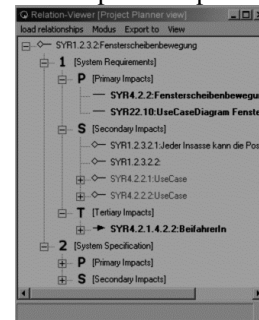


Figure 12. Project-Planner-View

Figure 12 shows the *project planner-view* of the *Relation-Viewer*. The figure visualizes an impact analysis for the requirement SYR1.2.3.2, which is shown at the top. The items with the name *System Requirements* and *System Specification* are requirements documents. Under these items, the change items are listed, ordered by the impact type.

Relation-Viewer offers the option to export the change impacts of an impact analysis to *Excel*, with all attributes, defined in *RequisitePro*. This enables the project planner to use the rich calculating functionality of *Excel* for estimating the costs of a change.

5. Improved activities

This chapter describes how our approach supports the different development and management roles investigated. Especially, it summarizes the answers to the questions the different roles have to tackle.

5.1. Requirements manager

We introduce a new role: the requirements manager. Task of this role is to prepare all tracing activities within an organization. Preparing tracing activities includes adapting our conceptual trace model for the organization, deriving relationship and tracing guidelines, and defining templates. For a comparable documentation structure defined in QUASAR [13], the defined trace model and

guidelines can be used. For another documentation structure, the trace model and the guidelines have to be adapted.

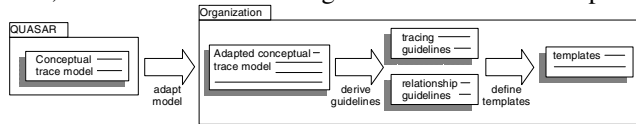


Figure 13. Working steps of the requirements manager

Figure 13 shows the working steps of the requirements manager. He adapts an existing conceptual trace model, e.g. the QUASAR trace model, and adapts it to the needs of his company. In a next step, he derives the guidelines and finally the templates.

5.2. Requirements engineer

The requirements engineer applies the relationship and tracing guidelines and uses the *QuaTrace* components *Rhapsody-Interface* and *Relation-Finder*. Figure 14 shows in which working steps of the requirements engineer our approach is used. A concrete working scenario is given in the following paragraph.

The requirements engineer created the “system requirements document” with support of *RequisitePro*. Now, s/he documents the new functionality *Func C*, in form of a natural language paragraph with *Word* and integrates a use case into a use case diagram with the help of *Rhapsody*. Then, s/he exports the use case with the help of the *Rhapsody-Interface* to *RequisitePro*. Then, s/he applies the relationship and tracing guidelines to establish traces between related requirements (e.g., a trace between the natural language paragraph and the use case that describe both the functionality *Func C* is established by using the same names). Then, s/he extracts traces (e.g., the name trace between the natural language paragraph and the use case) by using *Relation-Finder*.

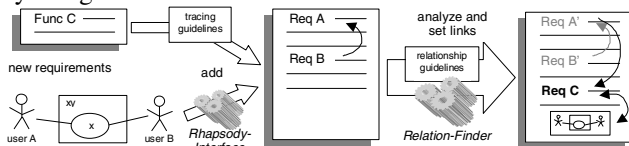


Figure 14. Improved working steps of the requirements engineer

The questions of the requirements engineer are answered by our approach as follows.

- UML elements are integrated into *RequisitePro* with the help of the *Rhapsody-Interface*.
- The granularity of the entities that should be traced is determined by the relationship guidelines.
- The traces that should be documented are described by the relationship guidelines.
- Guidance on how to document traces is given by the tracing guidelines.
- The effort for establishing links is reduced by an interface between the used tools. In addition, the tracing guidelines use implicit links and templates.

- Traces between documentation entities managed by different tools (e.g., natural language paragraph use case) can be established with the help of the *Rhapsody-Interface*.

5.3. Project planner

The project planner uses the *QuaTrace* component *Relation-Viewer* to analyze the impact of a proposed change (semi-) automatically. Figure 15 shows in which working steps of the project planner our approach is used. A concrete working scenario is given in the following paragraph.

The functionality *Func A* has to be changed. The project planner analyzes the corresponding requirement *Req A'* in the requirements document. For performing an impact analysis, s/he uses the tool *Relation-Viewer*. It offers him a fine-grained impact analysis, helping him rating the effort and costs for changing *Func A* more precisely. Finally, s/he estimates the costs for the planned change.

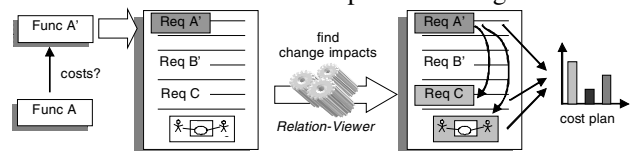


Figure 15. Improved working steps of the project planner

The questions of the project planner are answered by our approach as follows.

- Relationship and tracing guidelines should ensure that all traces are established and that all established traces are right
- (Semi-) automatic analysis to determine a set of change impacts
- Weight types of relationships differently and use this weight for cost estimation. Export functionality for Excel.
- The *QuaTrace* component *Relation-Viewer* weights different types of relationships differently. These weights are used for cost estimation. The *Relation-Viewer* exports change impacts and their weights to an *Excel* file.

5.4. Maintainer

The maintainer uses the *QuaTrace* component *Relation-Viewer* to identify further change impacts. In addition, relationships and tracing guidelines are used to update established traces and add new traces. Figure 16 shows in which working steps of the maintainer our approach is used. A concrete working scenario is given in the following paragraph.

The maintainer integrates an accepted change to *Func A*. The impact analysis of the project planner resulted in the change impacts *Req A'*, *Req C* and the use case. The maintainer checks whether these impacts are really affected by the change with support of the *Relation-Viewer* and identifies that only *Req A'* and *Req C* have to be

changed. When changing the change impacts with the help of *RequisitePro*, s/he analyzes and identifies a trace between *Req B* and *Req A* with the tool *Relation-Viewer*. Because of the modified interface of *Req A'*, *Req B* has to be changed, too.

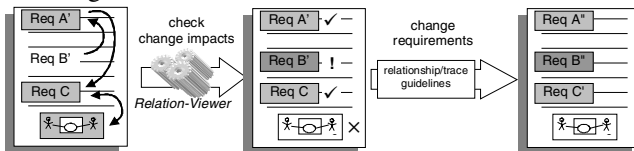


Figure 16. Improved working steps of the maintainer

The following analysis of related requirements of *Req B* has no results.

The questions of the maintainer are answered by our approach as follows.

- The relationship and tracing guidelines ensure that all important traces are established.
- The relationship and tracing guidelines ensure that all established traces are right.
- The *QuaTrace* component *Relation-Viewer* analyzes the traces automatically.
- The relationship and tracing guidelines as well as the different *QuaTrace* components ensure that the effort for maintaining traces is reduced.

6. Conclusion

We developed a fine-grained traceability technique and a supporting tool environment to establish, analyze, and maintain traces effectively and efficiently. So far, our approach provides support for the requirements engineer, project planner, and maintainer. The traceability technique was already validated in the domain of building automation. [10] shows how to derive guidelines for changing requirements and design documents based on a conceptual trace model. Experimental results showed a significantly beneficial influence of the guidelines on the correctness and completeness of a predicted set of change impacts in comparison to traditional development guidelines [6], [10]. In addition, we applied the traceability technique and a supporting DOORS implementation in an industrial context to support the recycling of requirements [7].

We see three major directions for future work to improve the traceability technique. All directions are tackled in ongoing research projects at the Fraunhofer IESE:

- *Extension to further software documents*
The conceptual trace model does not consider software documents, such as test cases or verification documents. Consideration of quality assurance documents will result in an extension of the conceptual documentation model.
- *Extension to further types of changes*
So far, our approach focuses on functional changes. The conceptual trace model does not describe how non-functional system characteristics are refined or realized.

- *Integration of cost models*

The impact analysis process from the viewpoint of the project planner results in a set of change impacts that can be used to estimate costs of required changes. The cost estimate depends on the underlying model that support estimating the costs based on the change impacts. It would be beneficial to combine our model with a cost model to get better estimates of costs. Feedback (i.e., comparison of the predicted with the actual change impacts and effort) can be used to improve the estimates.

Our tool environment supports the application of our traceability technique in a concrete project context. It can reduce the effort required for change activities. So far, we evaluated the tool environment only with examples. With the help of the tool, we managed different requirements documents that include textual requirements and models. We defined different types of change requests and implemented the changes with the help of the tool environment. The results are promising. We see three major directions for future work to improve the tool environment. All three directions are tackled in ongoing research projects at Fraunhofer IESE:

- *Empirical evaluation of the tool environment*

We assume that the main benefit of applying our tool environment is the efficiency of establishing, maintaining, and analyzing traces. The quality of the traces is improved mainly by the tracing guidelines. Therefore, we defined following research hypotheses for an empirical evaluation of the tool environment:

- Traces can be established and maintained with less effort than applying traditional tools (e.g., with *RequisitePro* and *Rhapsody*, without additional support)
- A complete and correct set of change impacts (from the viewpoint of the project planner and the maintainer) can be determined with less effort than applying traditional tools.

- *Support for naming consistency*

The requirements engineer uses name tracing for setting links. Typing errors and the use of wrong names have the effect that requirements will not be traced or will be traced wrong. So far, *QuaTrace* does not offer any consistency check for name tracing.

- *More support for filling out requirements documents*

So far, the requirements engineer has only the guidelines for filling out the requirements document. It would be beneficial to offer tool support to guide and check the filling out of the requirements documents.

Besides the tool environment *QuaTrace*, we developed two further tools that implement parts of the trace model and support different change activities. We integrated consistency checks in a commercial CASE tool (StP/UML). In addition, we developed a prototype that supports increment impact analysis and (semi-) automatic changing. These tools were investigated in two case studies. The results show that changes could be implemented with less effort.

In addition, the extended development tool allowed the verifier to perform a consistency checking that is more effective [6].

7. Acknowledgments

We would like to thank the working group of Software Engineering, especially, Dieter Rombach for his comments on previous versions of this paper. Furthermore, we thank the colleagues of the Fraunhofer Institute, especially, Barbara Paech. This work was partially funded by the German BMBF in the project QUASAR under grant VFB0004A.

8. References

- [1] S. A. Bohner and R. S. Arnold, "Software Change Impact Analysis", *IEEE Computer Society Press*, Los Alamitos, California, 1996.
- [2] M. Lindvall and K. Sandahl, "How well do experienced software developers predict software change", *The Journal of Systems and Software*, Vol. 43, 1998, pp. 19-27.
- [3] <http://www.rational.com/products/reqpro/index.jsp>
- [4] <http://www.ilogix.com/products/rhapsody/index.cfm>
- [5] B. Ramesh and M. Jarke, "Towards Reference Models for Requirements Traceability", *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, 2001.
- [6] A. von Knethen, "A Trace Model for System Requirements Changes on Embedded Systems", *International Workshop on Principles of Software Evolution (IWPSE' 01)*, 2001.
- [7] A. von Knethen, et al., "Systematic Requirements Recycling through Abstraction and Reuse", *RE 2002*, 2002.
- [8] M. Lindvall, "A Study of Traceability in Object-Oriented Systems Development", Licentiate thesis Linköping Studies in Science and Technology, No 462, Linköping University, Institute of Technology, Sweden, 1994.
- [9] D. Leffingwell and D. Widrig, "Managing Software Requirements. A Unified Approach", Addison-Wesley, 2000.
- [10] A. von Knethen, "Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems", PhD Thesis, University of Kaiserslautern, Germany, 2001.
- [11] <http://www.telelogic.com/products/doorsers/index.cfm>
- [12] M. Grund, "Verfolgung funktionaler Anforderungen mit Hilfe existierender Werkzeuge", Master Thesis, University of Kaiserslautern, Germany, 2003.
- [13] B. Paech, et al., "Integrated Quality Assurance and Requirements Analysis for the Software Development in Automotive Systems", <http://www.first.gmd.de/svt/quasar/english>, 2002.
- [14] B. Smith, "The Application of a Tool Like RDD-100 to Accommodate Traceability from Requirements to Test Results", <http://www.alc.com/archive.shtml>, 1999.
- [15] J. Börstler and T. Janning, "Traceability Between Requirements and Design: A Transformational Approach", *Proc. 16th International Computer Software & Application Conference*, 1992, pp. 362-368.
- [16] A.C.W. Finkelstein et al., "Inconsistency handling in multi-perspective specifications", *IEEE Transactions on Software Engineering*, Vol. 20, No. 8, 1994, pp. 569-578.
- [17] K. Pohl, "PRO-ART: A Process Centered Requirements Engineering Environment", In: M. Jarke, C. Rolland, A. Sutcliffe and R. Dömges (Publisher), *The NATURE of Requirements Engineering*, Shaker Verlag, 1999, pp. 255-278.
- [18] A. van Lamsweerde et al., "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Transactions on Software Engineering*, Vol. 24, No. 11, 1998, pp. 908-926.
- [19] K.B. Gallagher and J.R. Lyle, "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, 1991, pp. 751-761.
- [20] K. Pohl, "Process-Centered Requirements Engineering", John Wiley and Sons, 2nd Edition, 1996.
- [21] D. Parnas and J. Madey, "Functional Documentation for Computer Systems Engineering", *CRL Report 237*, McMaster University, Hamilton, Ontario, Canada, 1991.
- [22] R. Janicki et al., "Tabular Representations in Relational Documents", *Technical Report CRL 313*, Telecommunication Research Institute of Ontario (TRIO), Nov. 1995.
- [23] D.P. Kirkman, "Requirements Decomposition and Traceability", *Journal Requirements Engineering*, Vol. 3, No. 2, 1998, pp. 107-114.
- [24] C.V. Ramamoorthy et al., "The evolution support environment system", *IEEE Transactions on Software Engineering*, Vol. 16, No. 11, 1990, pp. 1225-1234.
- [25] B. Ramesh and V. Dhar, "Supporting Systems Development by Capturing Deliberations During Requirements Engineering", *IEEE Transactions on Software Engineering*, Vol. 18, No. 6, 1992, pp. 498-510.
- [26] INCOSE Requirements working group, "Requirements Management, Requirements Analysis, and Requirements Engineering methodology", <http://www.incose.org/tools/>.
- [27] X. Song et al., "Lessons Learned From Building A Web-Based Requirements Tracing System", *Proc. of International Conference on Requirements Engineering*, 1998, pp. 41-50.
- [28] F.A.C. Pinheiro and J.A. Goguen, "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, Vol. 13, No. 2, 1996, pp. 52-64.
- [29] O.C.Z. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem", *Proc. International Conference on Requirements Engineering (ICRE)*, IEEE CS Press, 1994, pp. 94-101.
- [30] IEEE Std. 610.12-1990, "Standard Glossary of Software Engineering Terminology", 1990.
- [31] R.J. Turver and M. Munro, "An early impact analysis technique for software maintenance", *Journal of Software Maintenance – Research and Practice*, Vol. 6, 1994, pp. 35-52. Also published in [1].