Defining a Traceability Link Semantics for Design Decision Support

Robert Brcina and Matthias Riebisch

Technical University of Ilmenau, Germany
robert.brcina|matthias.riebisch@tu-ilmenau.de

Abstract. The development and the evolution of large, complex software systems bear several risks. Traceability links can help to master the complexity of these tasks. Currently, they are not used in a large scale, because tool support is necessary to reduce the overhead effort. At present, tools for handling traceability links cannot be effectively developed, because the syntax and the semantics of the traceability links are not sufficiently defined. In this paper we present a set of traceability link types together with a definition of their semantics. The set of link types was developed by analyzing the link evaluation and exploitation. The presented link types are customized for the support of architectural design decisions in regard to a set of non-functional design goals. The extension of the results to a wider scope is discussed. The work was performed within a large industrial project.

1 Introduction

Large, business critical software systems have to perform a tough succession of changes in order to maintain their value for a company. Changes of complex software systems bear several risks. Design decisions have to be made under uncertain conditions, because the consequences of different alternatives cannot be determined precisely. Traceability can support the decision making by facilitating the software comprehension, the change impact analysis, and the minimization of risks. However, the accuracy of the traceability links constitutes a critical issue to achieve the expected benefit. If the information provided by these links is wrong, bad decisions and the introduction of errors are a consequence.

For maintaining the accuracy of the traceability links, tool support during the establishment, the adaptation and the evaluation of the links is necessary. Furthermore, the link maintenance by humans requires a high effort and introduces new risks of mistakes. Currently, an effective tool support can hardly be provided because the semantics of the artefacts and models used, and of the traceability links themselves is not defined precisely enough. Guidelines and rules for link modifications during changes and for evaluations are only provided as far as the semantics of the links and of the linked artefacts are defined. Furthermore, there is a broad variety of proposals for link types, but little attention has been paid on the definition of the link semantics. The semantics has to support the link utilization in order to be of practical value. A pure categorization of links

 ISBN 978-82-14-04396-9

constitutes an important step, but it does not lead to the required preconditions for a tool support.

In this paper we present a traceability link definition framework for supporting architectural design decisions. Traceability links have to represent relations between artefacts in different phases of the development process. Depending on the goal of the decision, different types of artefacts and relations have to be considered. Due to space limitations, we focus on architectural design decisions regarding the non-functional design goal evolvability. We start from a subset of the currently used link types and define their semantics by the way of their utilization. Even if the results of this work are applicable only to these decisions, we expect that the discussion about link types and semantics is driven forward and that they lead to a significant support for the tool development.

After a brief discussion of related work, section 3 introduces the traceability approach. Based on them, in section 3.3 the indicators and the corresponding resolution actions are introduced. In section 3.4 the application of the approach in an iterative development process is illustrated by an example. In section 4 we introduce the link meta model and the link semantics.

2 Related Work and Traceability Link Utilization

From the engineering point of view *traceability links* are used to trace design decisions during the development process. Both, *functional* and *non functional tracing* allow following functional and non functional issues of the system development [10]. They facilitate system comprehension by providing the required information about relations between artefacts and entities, e.g. the scenario based approach described in [7]. A *traceability model* is used to define the required entities and relations during the software development, e.g. in [8]. The definition of relations as traceability link types is important for the utilization of the model information. Unfortunately, the definition of a standard set of traceability link types is still an unresolved issue. However, for a tool support of design evolution a semantic differentiation of the traceability link types is needed. Due to different research goals, a high number of traceability link type definitions has been established, e.g. in [9] or [8]. As a step toward simplification and abstraction, we will later restrict ourselves mostly on the *implementedBy* traceability link type plus the dependency relations of the modelling language UML2.

Traceability Link Utilization. Link types should be defined in a way suitable for the intended usage. In the following we list a set of activities in which the links are used, together with the goals of using them.

Verification of (forward) engineering activities: identification of the input to an engineering activity (e.g. requirements, goals, models, risks); understanding and making a decision; verification of the completeness of an activity; verification of the design rules applied.

Change impact analysis: identification of all entities depending on the changed one; understanding the type of dependency to a related entity in order to identify the necessary way of changing it, accordingly.

ISBN 978-82-14-04396-9

Software comprehension and reverse engineering: identification of all related entities to the one in focus; understanding the type of relation between the entity in focus and a related one; identification of abstractions, e.g. design patterns, architectural styles, principles.

Identification of the source of a decision or requirement: identification of the stakeholder who demanded a particular property; justification of a decision effort; resolution of a set of contradicting requirements.

Decision support: understanding the influence factors and the goals of a decision; establishment of proposals for solutions; evaluation of alternative solutions.

System configuration and versioning: identification of constraints between components; identification of necessary changes to resolve a constraint; identification of differences between two versions of the same artefact and their impact on other ones.

3 Traceability Approach for Design Decision Support

3.1 Architectural Design Decisions for Evolvability

Decision-making and assessment are both critical activities for the success of development processes because they apply the success criteria. They are performed during and after work on artefacts. Assessments have to be performed as early as possible to provide early feedback for developers and to minimize rework. They provide the means to control iterative development processes.

Elaboration of the criteria for the assessments. In the following, we will apply evolvability as one criteria for architectural quality in long-term development projects. It is influenced by an appropriate use of the concepts of abstraction, delegation, modularization [4], conceptual integrity [5] and separation of concerns [6]. Beside these aspects, there are additional ones related to general issues of software development processes, such as the availability of a proper set of documentation. We will focus on the concepts of modularization, encapsulation and separation of concerns for the assessment of the goal evolvability. Additionally, the ease of change at architectural level is an important criterion. Problems arise from effects called scattering, tangling and insulated artefacts, which hamper the above mentioned criteria and the quality attributes. Based on a traceability approach and on the set of defined traceability links we introduce indicators (see section 3.3), which enable us to analyze, reveal and reduce these effects.

Artefact Categories. In our approach, we consider relations between requirements, architectural elements and implementation. The key idea is to enable a tracing of all software elements back to the requirements. According to the method used for the system development, different artefacts are involved. Within a feature driven development we consider four types of artefacts: features (F), architectural components (A), classes (C) and implementation artefacts (I). Examples for I type artefacts are configuration units.

Typical effects. From a point of view of changeability, a 1:1 relation between dependent objects is the most effective one (Fig. 1 left). In such a case the change

ISBN 978-82-14-04396-9

of a feature f_0 requires only the change of the precisely related component a_0. Following all related objects, the ideal but usually not realistic case is that each object does not have more than two traceability link connections. In this case the alignment of components to features is possible. It enables minimal invasive changes, as the features can be exchanged by code composition.

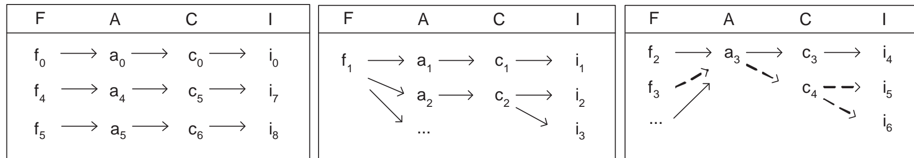


Fig. 1. Ideal case (left), feature scattering (center), feature tangling (right).

In practice, more dependencies have to be considered. A high number of dependencies means that more artefacts are affected by a change resulting in a higher maintenance effort and a reduction of the variability. The two important types of effects are discussed using Fig. 1. Feature scattering (Fig. 1 center) means that one feature f_1 is implemented by more than one architectural component – in this case the components a_1, a_2 and others. In the case of feature tangling, an architectural component is responsible for more than one feature (Fig. 1 right) e.g. the implementation of feature f_2, f_3 and others is tangled in component a_3. If one of these features has to be removed, the component a_3 has to be analyzed and split into appropriate parts, with a much higher effort than just the removal of one component. In order to improve the flexibility of a system concerning feature variability, all variability points should be aligned in a way that each of them is related to exactly one optional feature.

Corresponding artefacts depending on the evolution of one feature, e.g., system components and classes are related by traceability links. The right part of Fig. 1 shows an example: the class traceability enables a discrimination between the relation of c_3 and f_2 (indicated by a solid arrow) from the one between c_4 and f_3 (dashed arrow). An impact analysis for feature-related changes is facilitated by this discrimination.

3.2 Types of traceability links and artefacts

In the following, we will show a cutout from an industrial IT infrastructure project for illustration purposes. For the chosen cutout it is not necessary to consider implementation artefacts. The set of features F and the considered subset $\mathcal{F} \subseteq F$ both are contained in the feature model. The set of architectural software components A and the considered subset $\mathcal{A} \subseteq A$ are part of the architecture model. The set of classes C and the considered subset $\mathcal{C} \subseteq C$ are part of the realization (design) model. The traceability links, which are required for the later evaluation by the indicators (section 3.3) are defined in Table 1.

 ISBN 978-82-14-04396-9

Traceability Category	Traceability-Link-Type	Link-Source	Link-Destination	Link-Symbol
Component Traceability	*implementedBy*	Feature	Component	$f \rightsquigarrow \mathcal{A}$
Class Traceability	*implementedBy*	Feature	Classes	$f \rightsquigarrow \mathcal{C}$
Component Require Relation	*use*	Component	Component	$a \mapsto \mathcal{A}$

Table 1. The used set of traceability links.

Component Traceability. Each component contributes to a set of requirements. Such a relationship is expressed by the *implementedBy* traceability link pointing to components that implement a set of features.

Class Traceability. Software components consist of a set of classes and vice versa a class c is related to exactly one software component in order to implement at least one part of a feature. For classes the same traceability link type *implementedBy* is used as for features and components.

Component Require Relation. Similar to class traceability this kind of use of traceability links describes the relationship between two components in which one component needs the others to implement the related feature.

3.3 Metrics for evaluation: Scattering and Tangling

The above defined traceability links are used to establish indicators – often called metrics – which enable us to evaluate architectural design decisions regarding the quality attributes for evolvability. The indicators are accompanied by actions for problem resolution and explained in the following, whereas section 3.4 illustrates their application during the evolutionary development. Due to the space limitation, the insulated features effect cannot be discussed here. The traceability link based indicators defined here, together with a variety of other indicators [11] are applied for design decision support and architectural evaluation.

Feature Scattering. Feature scattering affects the evolvability of a system because the change of one feature demands changes of more than one components, thus leading to higher effort and to a higher probability of mistakes than in the ideal case. On the architectural level, feature scattering refers to a relation between one feature and more than one components. In order to avoid a division by zero while calculating the feature scattering indicator, all insulated features and insulated components have to be removed before. The traceability link type necessary for the indicator is indicated within the definition using the traceability link symbol defined in Table 1.

Definition: Feature Scattering Indicator
$fsca$ is based on $a \in A$, $f \in F$ and is defined as follows:

$$sca\,(f) := |\{a : f \rightsquigarrow a\}| - 1, \text{ and} \tag{1}$$

ISBN 978-82-14-04396-9

$$fsca\,(F) := \frac{\sum_{f \in F} sca\,(f)}{|F| \cdot |A|},\ fsca \in [0,1)\,. \tag{2}$$

The more features are scattered into components, the worse the evolvability of the software gets and the closer the result of the indicator moves to 1. The maximum value of 1 is reached if $|a|$ approaches infinity and each feature $f \in F$ is implemented by all $a \in A$.

Resolution: Reducing feature scattering

A reduction of the feature scattering could be achieved (a) by splitting up the features into several ones starting with the feature with the highest value for $sca\,(f)$, (b) by merging components to reduce the number of involved components.

Feature Tangling. Feature tangling refers to relations between more than one feature and one component. In order to avoid a division by zero while calculating the feature tangling indicator, all insulated features and insulated components have to be removed before.

Definition: Feature Tangling Indicator
The Feature Tangling Indicator $ftang$ is defined as follows for $a \in A$:

$$tang\,(a) := |\{f \in F : f \rightsquigarrow a\}| - 1, \text{ and} \tag{3}$$

$$ftang\,(A) := \frac{\sum_{a \in A} tang\,(a)}{|F| \cdot |A|},\ ftang \in [0,1)\,. \tag{4}$$

The more features are tangled to one component, the more difficult is the adaptation of this component and the closer is the result of the indicator to one. The maximum value of 1 would be reached if $|f|$ would approach infinity and each component $a \in A$ would implement all features $f \in F$. An example for $ftang$ is shown in section 3.4.

Resolution: Reducing feature tangling

A reduction to the feature tangling effect could be achieved by (i) splitting up the component starting with the highest number of $tang\,(a)$ into several components, each with a reference to the corresponding feature, or (ii) by merging features to reduce the number of involved features.

3.4 Illustrating Example

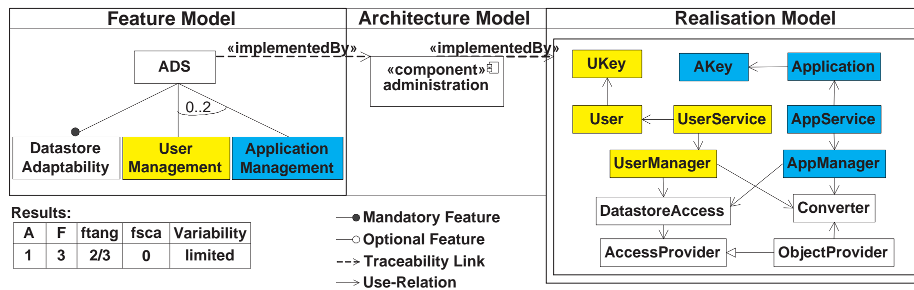
We illustrate the application of our traceability approach by evaluating the architectural solution of an Administration System (ADS) part of an IT-Infrastructure by using the indicators defined in the previous section. We will examine one development iteration (we call it evolution step) and its rework. The goals are evolvability and the ease of change on architectural level in order to support

ISBN 978-82-14-04396-9

variability. Due to space limitations only a part of the IT infrastructure project is visible in the example.

Evolution Step. All three features *User* and *Application Management* and *Datastore Adaptability* are implemented at the levels of features, architecture and classes. The feature *Application Management* allows to manage all application specific information, whereas the feature *User Management* allows to manage all user specific information of the IT infrastructure. As shown in Fig. 2, traceability links are used to express the dependencies between these models.

Evolution Step



Rework of the Evolution Step

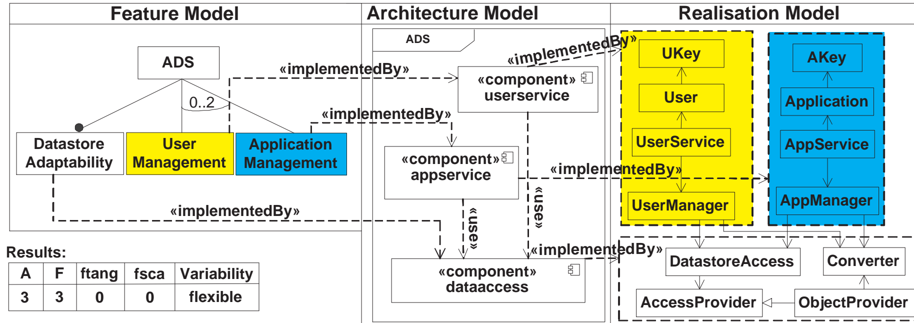


Fig. 2. The evolution step (above) and its rework (below).

Change operations within each evolution step are recorded by traceability links. A feature tangling effect is revealed by three traceability links starting from each feature to the component *administration*. As defined in Table 1 there are two types of traceability links, *use* between components and *implementedBy* links between features and components as well as classes. Additionally, relations between classes are considered.

The evaluation results using the indicators are summarized in Fig. 2 in small tables at the lower left. The result $ftang = 2/3$ indicates that there is a tangling between the features and the architectural component, which hampers the evolvability of the ADS system. In addition to evolvability it is important to achieve a high flexibility regarding changes of features. To support variability

ISBN 978-82-14-04396-9

at architectural level, the components shall be used independently; and the customers can select a random set of optional features, e.g. $User\ Management$ or $Application\ Management$ and their combination. The application of the indicators reveals that the system has a limited variability because of the feature tangling: the two features $User$ and $ApplicationManagement$ have to be deployed even if only one is needed. In this case the component *administration* consists of classes from both features, as indicated in Fig. 2 by corresponding shades of grey. With the resolution actions these limitations will be resolved by the following rework.
Activities for improvement. The tangling has to be eliminated by splitting the component *administration* into three architectural components. The traceability links indicate that a decomposition into components related to features is possible. The result of the resolution actions is presented in the lower left of Fig. 2. Comparing the results of the evolution step with those of the rework shown in the tables, we state a resolution of the feature tangling from $ftang = 2/3$ down to $ftang = 0$. As a result of the rework it is possible to use all optional features without an overhead effort for configuration. However, the success with this variability goal causes a dependency between the components.

4 Development of a Link Meta Model and a Link Semantics for Design Decision Purposes

The definition of the traceability link semantics - together with the definition of a metamodel - has a big influence on the resulting overhead effort for the link maintenance and management. Following the goals of effort minimization, the definitions are as lightweight as possible. This leads to link semantics with as few as possible, but as many as necessary aspects covered.

We have discussed only a very small subset out of the bandwidth of traceability link utilizations as mentioned in section 3. However, we are able to give application details within the space limitations of this paper. Even if the established link metamodel will have a limited scope, our procedure provides an example for the development of traceability link frameworks based on the intended ways of utilization.

For the design decision support regarding evolvability we only need links of the types *implementedBy* and *use*. As explained in section 3, only links between artefacts of the type feature, component and class are evaluated for a calculation of the indicators. Information about the source and the destination of each link is sufficient for this purpose. For the establishment and the evaluation of different alternatives for design decisions, additional traceability links are required to evaluate the priority of competing design goals. These links connect components and classes to features and are of the type *implementedBy*. As additional information, these links carry design decisions about the way in which influence factors are considered. During the mentioned activities for improvement, new traceability links are established however they do not cover more information or additional types than already mentioned.

ISBN 978-82-14-04396-9

The necessary information for the traceability link can be represented in a metamodel definition. Due to the limited scope we expect less information than mentioned in works with a broader scope, e.g. our earlier work [2]. The resulting definition is expressed using UML as a metamodel as shown in Fig. 3.

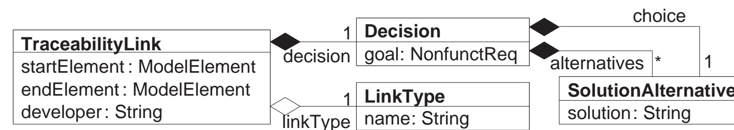


Fig. 3. Traceability Link Metamodel.

The semantic information is covered by the definitions of the link types and artefact types, as well as by the rules which apply to the links. We can distinguish rules at different levels: regarding context-free syntax, context-sensitive syntax, static and dynamic semantics. Rules regarding dynamic semantics are not considered so far. As examples, we mention a few rules related to syntax:

1. Each feature is related to one or more components by an *implementedBy* link.
2. Each component is related to one or more classes by an *implementedBy* link.
3. Each component is related to one or more features by an *implementedBy* link.
4. Each class is related to one or more components by an *implementedBy* link.

The rules for link semantics express design rules. We can distinguish rules of different categories: (a) Very general rules representing general engineering principles, e.g. decomposition and abstraction, (b) method-specific rules, e.g. a rule about the mapping of non-functional goals to functional solution principles according the architectural method by Bosch [1], and (c) domain-specific rules, e.g. a rule which type of response is valid for a certain type of event in a specific telecommunication protocol. We have to state, that only a few rules have criteria which are precise enough to enable clear statements. They can be implemented in tools for an automatic evaluation and verification of models. Most of the precise rules belong to the syntax-related ones. Unfortunately most semantic rules cover less strict criteria; therefore they provide only degrees of fulfillment between true and false. They can be used for human inspection only, but they provide valuable hints and enable a reduction of the search space. Therefore they increase the efficiency of an inspection by reducing the effort and by enabling a concentration and an increased precision.

We just want to mention that there is another type of rules frequently called heuristics which have even less clear criteria. They are used for other ways of link utilization e.g. for impact analysis which are out of the scope of this paper. They are applied e.g. for controlling how far links are tracked and to what level of detail links are maintained, depending on an actual risk.

ISBN 978-82-14-04396-9

5 Conclusion and Future Work

In this paper we have shown the application of traceability links for the support of architectural design decisions. Evolvability was considered as an example of a quality property of architectures. By an example it was shown how links have to be defined in terms of syntax and semantics, to provide the best support for architectural decisions. Even if the purpose is in some way specific, the procedure can be expanded to further ways of link utilization.

The idea of model-based development behind our research aims the coverage of all necessary information in models and the used traceability links. One could dispute that this leads to heavy-weight development processes with a high modeling effort, but a strong restriction to the necessary parts of information and an exhaustive utilization of the models helps to increase the overall efficiency. The next steps of our work include the investigation of the necessary adoptions for other ways of link utilization including a refinement and a revision of the defined link semantics, the evaluation of applicability of these link definitions for the other utilization activities, and the extension of the rule set during empirical research. The link definitions represent a prerequisite to the development of a comprehensive tool support to provide a (partly) automated link management.

References

1. Bosch, J.: Design and Use of Software Architectures – Adopting and evolving a product-line approach. Addison-Wesley(2000)
2. Maeder, P., Philippow, I., Riebisch M.: Customizing Traceability Links for the Unified Process. QoSA 2007, USA, 2007. Springer LNCS, pp. 47-64 (2007)
3. Riebisch, M.: Supporting Evolutionary Development by Feature Models and Traceability. Proc. ECBS2004, IEEE Computer Soc., pp. 370-377 (2004)
4. Parnas, D. L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12, 1053-1058 (1972)
5. Brooks, F. P.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley (1995)
6. Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. Commun. ACM vol. 44, 10, 43-50 (2001)
7. Egyed, A.: A Scenario-Driven Approach to Traceability. ICSE'01, pp. 123-132 (2001)
8. Ramesh, B., Jarke, M.: Toward Reference Models for Requirements Traceability. IEEE Trans. Softw. Eng. vol. 27, 1, pp. 58-93 (2001)
9. Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability. ICRE'96, IEEE Computer Soc., pp. 76-84 (1996)
10. Pinheiro, F. A. C.: Requirements traceability. In Requirements traceability in Perspectives on Software Requirements. Julio C. S. P. Leite and Jorge Doorn, Kluwer Academic Publishers, pp 91-113 (2004)
11. Matthias Riebisch and Robert Brcina: Optimizing Design for Variability Using Traceability Links. ECBS, IEEE Computer Society, pp. 235-244 (2008)

ISBN 978-82-14-04396-9