Traceability & Mega-modeling
An introduction for undergraduates

Maximilian Meffert

University of Koblenz-Landau

Abstract. Traceability and mega-modeling deliver a strong foundation to comprehend and work with big systems but neither of those concepts is part of an undergraduate curriculum in computer science. However graduation qualifies to work with even large systems. This paper introduces both ideas on an undergraduate level and exemplifies their beneficial combination by comparing three approaches alongside their targeted development phases.

Keywords: traceability, mega-modeling, requirement engineering, model driven engineering, architecture frameworks, runtime models

1 Introduction

Software and computer systems tend to become bigger and more complex to meet higher expectations. At the same time the complexity of engineering processes creating and maintaining such systems increases proportionally. One way of taming this rising complexity is the concept of *traceability* which is commonly interpreted as the system ability *"... to describe and follow the life of software artifacts ..."* [8], but it is not yet part of an undergraduate curriculum in computer science. Neither is *mega-modeling* due to its high level of abstraction, as it deals with models using other models as their elements. We argue that it is reasonable for undergraduates to get familiar with both concepts regarding the fact that an achieved degree qualifies to work with even large systems.

In this paper we will outline the basics of traceability and mega-modeling on an undergraduate level, and show how the combination of both can be beneficial to engineers in terms of *system understanding* and *process management automation* so mega-modeling allows one to use model driven engineering (MDE) techniques and technologies.

1.1 Contributions of this paper

Contributions

Non-Contributions

1.2 Road-map

§2 introduces the concept in terms of terminology and motivation of traceability. §3 introduces the concept of mega-models. §**??** revisits traceability objectives and introduces a higher categorization. §4 summarizes three approaches on traceability, mega-models and traceability mega-models. §5 concludes the paper.

2 Basics of traceability

Historically the research on traceability in software engineering originates from requirement engineering (RE). The question is: given a discrete set of requirements, how can one validate or even prove that all requirements are met? And more importantly on what information can one base such validation?

The idea to solve this problem came through observation on typical product development processes where a raw vision is transformed to concrete requirements, requirements are transformed to architecture and architecture is transformed to a final implementation. Due to the incremental and iterative nature of such processes it is reasonable to assume that engineering activities might leave *traces* which reflect the performed activity and the past state of the modified artifact.

Later traceability became of more interest to the MDE community. The purpose of traceability here did not much differ from its purpose in requirement or software engineering. We just change our perspective on how we see and describe engineering processes. Usually we try to regard artifacts as models of some kind at best specified by meta-models, and activities are seen as also well defined model-transformations. On top of that MDE is strongly interested in the automation of engineering processes.

Winkler and von Pilgrim compiled a complete survey on traceability where origins and differences in both fields can be explored in more detail [6]. This section is strongly based on their work.

2.1 Traceability definitions

Traceability is subject of various research fields, hence there is no standard definition on what traceability actually is. The introductory citation is from Lago et al. [8] and states the following: *"Traceability is the ability to describe and follow the life of a software artifact and a means for modeling the relations between software artifacts in an explicit way"*. This definition is a good starting point if one wants to grasp what traceability is about because it is based on the life-cycle of artifacts and their relationships which has a nice intuitive notion. However, there is a weakness, it is not clear whose ability it should be. Other more technical definitions of traceability are given by the IEEE [2]:

1. *The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another.*

2. *The degree to which each element in a software development products establishes its reason for existing.*

Here, traceability is not seen as an ambiguous ability, on the contrary it is stated as a hopefully measurable degree of relationships. Moreover in definition 2 product elements are made accountable for their own traceability.

traces Vital to traceability is the idea that engineering activities leave traces. Such traces may appear in form of meta-information describing hard facts like *what* happened, *who* did something or *when* something happened. Opposing that traces also may appear as *"... (non-) material indication or evidence ..."* [13] on *why* or *how* something happened.

For example version control systems automatically record the user, modification date and the modification itself committed for a given artifact. Additionally a commit-message is passed alongside which hopefully gives some insights on the stored changes. These messages could refer to a customer interview, a discussion under colleagues or technical research to establish some reason on the modification. Such conversations are usually not transcribed, hence they are considered soft traceability information. Note: the modification itself is also considered to reflect such non-material traceability information, although this data is hard to obtain unless there is explicit documentation like sufficient doc-comments or commit-messages.

Following this separation in hard and soft fact traces Pinheiro [10] defines a classification which somewhat resembles the separation commonly used on requirements:

- **Functional traces:** Traces existing due to well defined transformations. Such transformations produce traces as by-products like revision numbers in version control systems. Or traces that may be obtained by analyzing the transformation input, output and rules. In short, these traces have to obey some sort of formalism.
- **Non-Functional traces:** Traces which cover *"reason, context, decision, and technical"* aspects. I.e. explaining the informal interpretation of MUST and SHOULD in requirements.

traceability links The IEEE [2] defines a trace as a record *"To establish a relationship between two or more products of the development process; for example, to establish the relationship between a given requirement and the design element that implements that requirement."*. The frequent notion of relationships describing or supporting traceability leads to the concept of *traceability links*. These links can be seen as a more technical description of traces as they are stated to be instances of n-ary, multi-directional relations, although they are not named by the IEEE Glossary like traces and are often used as synonyms for one another [6]. This can be misleading in environments which are not not strongly modeled where traceability links seem to not cover simple meta-data.

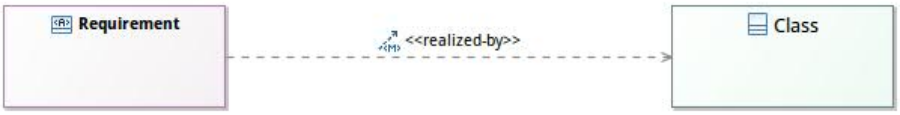

Fig. 1. Requirement realization link

A simple example for traceability links is shown in Figure 1 where the realization of a requirement by a class is denoted. However, this is a minimal example, in real-life the relation might need a higher arity to add constrains and test cases in order to confirm realization.

There are also categorizations on traceability links mostly considering the phase or abstraction nature. For requirements engineering Gotel and Finkelstein use a big phase classification differentiating between *pre- and post- requirement specification traceability* [11]. On the other hand Ramesh and Edwards distinct between *horizontal and vertical traceability* [12]. Figure 1 exemplifies vertical traceability as realization is a relation between abstraction layers.

Similar to these classifications there is *pre-, intra- and post model traceability* by Paige et al. for MDD considering traceability between the first non-model artifacts and the first model, traceability during model-refinement and traceability between the final model and generated artifacts [6].

Till now we introduced the mere concept of traceability which is far from applicable. To make actually use of traces and traceability links we need do define schemes in RE or meta-models in MDD which also may vary with their targeted domain. Traceability meta-models will be covered in a later section.

2.2 Traceability objectives

Another thing we like to point out besides the introduction to the rather theoretical concept of traceability is the motivation behind all this. Winkler and von Pilgrim [6] identified several objectives where traceability might be of great help. Figure 2 depicts traceability objectives sorted by the research field they originated from. However most objectives found in requirements engineering can also be considered important to the MDD community. Additionally some objectives already correspond directly, i.e. *Estimating change impact* and *Change impact analysis* do not differ at all regarding their purpose. Given sufficient traceability information in form of traces and traceability links one can simulate changes top-down to the final model and estimate their impact on metrics like quality, performance, cost, etc. The only difference between RE and MDE here lies in the propagation of changes. Where in RE changes are considered to be propagated manually, MDE uses automation. Although nowadays RE also relies on tool support.

Other corresponding objectives are *Proving system adequateness/ Validating artifacts* and *Proving adequateness/Validation*. If end-to-end traceability from early artifacts to the final implementation is achieved, one cannot only use traces

RE Traceability Objectives	MDE Traceability Objectives
• Prioritizing requirements • Estimating change impact • Proving system adequateness • Validating artifacts • Testing the system • Supporting special audits • Improving changeability • Extracting metrics • Monitoring progress • Assessing the development process • Understanding the system • Tracking rationale • Establishing accountability • Documenting re-engineering • Finding reusable elements • Extracting best practices	• Supporting design decisions • Proving adequateness/Validation • Understanding and managing artifacts • Deriving usable visualizations • Change impact analysis • Synchronizing models • Driving product-line development

Fig. 2. Traceability Objectives identified by Winkler and von Pilgrim [6]

to identify incompleteness (again top-down), it is also possible to check for requirement coverage (bottom-up). This is especially beneficial for the use case to attest customers that the system specification is met and not exceeded.

Besides these product centric objectives which mainly address maintainability and soundness concerns of a completed project, there are also objectives to support the development process itself. Such are *Monitoring progress* and *Establishing accountability* where project management can leverage end-to-end traceability to determine if a milestone is reached or who to blame if not.

Eventually we can learn from collected traceability data during a project revision while *Extract Best Practices* to support future design decisions.

3 Basics of mega-modeling

The second topic of current research on which we need basic knowledge to understand the coming sections is mega-modeling. The term mega-model is somewhat ambiguous as it commonly refers to models using other models as their elements. But the term model is not used in a restrictive manner [5], it may be interpreted as intuitive models, elements of such models, meta-models describing such models and moreover the term may be used for mega-models itself as depicted in Figure 3, so mega-models could actually describe mega-mega-models or even mega*-models. And to make things a little bit more confusing mega-models may also be defined by meta-models.

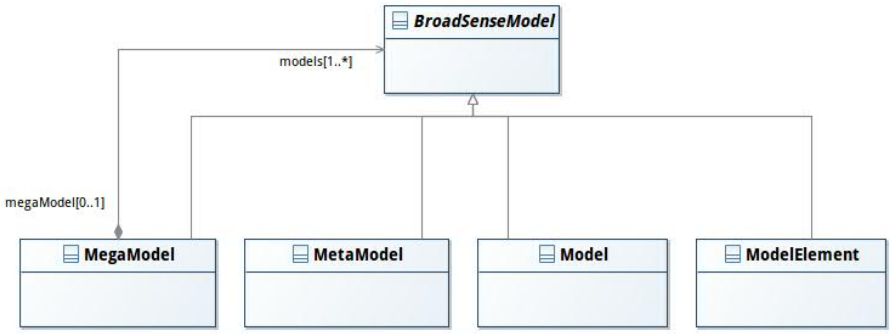

Fig. 3. Mega-models in a broad sense

3.1 Mega-model usage

Unsurprisingly mega-models are meant to be used for large-scale modeling, i.e. models of big enterprise systems [9]. Such systems are ecosystem-like environments consisting of various different technologies interacting with each other. Especially regarding this aspect of systems one can make use of mega-models to investigate the dependencies of involved *technological spaces* [3][5][9]. However the use of mega-models may not only be indicated by system size. Even minimal static [1] web-pages can use a multitude of technologies in form of

- software languages like HTML, CSS and JavaScript (nowadays CSS also may conform to corresponding LESS or SASS code),
- software APIs likes JQuery, Angular.js, etc.,
- infrastructure components like server and client/browser,
- and protocols like HTTP(S) or FTP.

Enhancing the web-page with server side logic might add the technological spaces of PHP-ware, Java-ware and/or SQL-ware.

Another use case or need for mega-models arises implicitly when one wants to build MDE supporting tools. MDE-tools provide functionalities to create and maintain digital representations of models, meta-models and model- transformations. To do so, such tools need to implement a meta-model capable of these three (ore more) aspects, whose object instances would in fact be some sort of mega-models. Seibel et al. [1] created a prototype which is able of managing traceability links between models and model-elements by utilizing a meta-mega-model.

3.2 The MDE-Mega-Model

One particular mega-model we want to point out is the MDE-Mega-Model (Figure 4) by Favre et al. [9].This approach aims to model MDE evolution processes.

[1] static in a sense that content data is not provided dynamically, i.e. form of databases

Although it is not yet complete, it already proves to be a powerful help analyzing technological spaces.

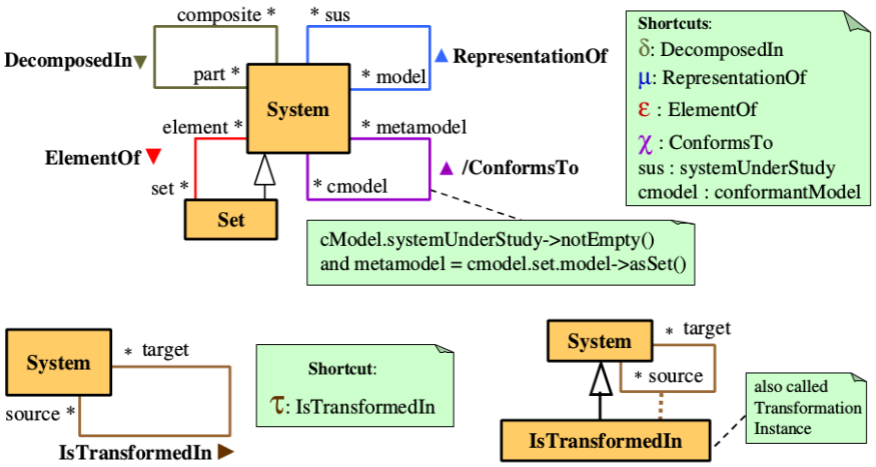

Fig. 4. The MDE-Mega-Model by Favre et al. [9]

The other reason to introduce this mega-model here is due to its concise nature and high level of abstraction regarding MDE. This makes it an ideal framework for analyzing other concrete mega-models.

Favre et al. identify five relations concerning systems which we will now explain in short:

systems At first we need to clarify the term *system*. Favre et al. state: *"A system is the primary element of discourse when talking about MDE"*, although the terms usage may not be important as everything can be seen as some kind of system [9]. This is exemplified with the trigonometric system π (meaning the real number). For the sake of simplicity and not repeating examples we may just think of a system as the *thing of interest*.

δ (`DecomposedIn`) Decomposition is a structural relationship which denotes that a *composite* can be `DecomposedIn` a *part*, or in set-theoretic notation:

$$composite\ \delta\ part \text{ or } (composite, part) \in \delta$$

Example: A directory in a file system tree contains simple files and other directories, so its safe to say: `dir` δ `file` and `dir` δ `dir`. The latter also illustrates the recursive notion of this relation.

μ (`RepresentationOf`) Representation is a descriptive relationship denoting a *model* as a `RepresentationOf` of a *system under study*. We write:

$$model\ \mu\ sus \text{ or } (model,sus) \in \mu$$

Again, we do not care that much about the term *system*, for *model* on the other hand we think of it as abstraction and/or simplification of such systems. Example: As directories contain n files, a list of files can be the representation of a directory: `[file0,file1,...]` μ `dir`.

ε (`ElementOf`) This is simply the set-theoretic $\in$ relationship, so an *element* is `ElementOf` a *set*. In greek notation:

$$element\ \varepsilon\ set \text{ or } (element,set) \in \varepsilon$$

Example: `FOO := (/foo)+` may be the set of all Unix-files following the this pattern (`FOO = { /foo, /foo/foo, /foo/foo/foo, ... }`), so `/foo` ε `FOO`.

χ (`ConformsTo`) Conformance adds the notion of meta-models, where a *conformantModel* `ConformsTo` a *metamodel*.

$$cmodel\ \chi\ metamodel \text{ or } (cmodel,metamodel) \in \chi$$

Example: `(/foo)+` is a model for `FOO`, `/foo(/foo)*` is also one, so we have `(/foo)+` μ `FOO`. Both are regular expression describing the same set of strings and thus have to obey the syntax rules for regular expressions (`RegExp`), or in the notation of this mega-model: `(/foo)+` χ `RegExp`.

τ (`IsTransformedIn`) Transformations play a very important role in MDE because the benefit of just being able to statically describe systems is limited. Additionally we need the ability to model the progress of a development process. But this is relatively simple to achieve by applying the well known concept of functions to this mega-model. So we may say a *source* `IsTransformedIn` a *target*, we think *source* $\mapsto$ *target* and write:

$$source\ \tau\ target \text{ or } (source,target) \in \tau$$

Example: One transformation class of particular interest are model transformations. Given the established model `(/foo)+`, we might have it transformed to the model `(/bar)+`. This is denoted as `(/foo)+` τ `(/bar)+` or (`(/foo)+`, `(/bar)+`). The latter is also called *transformation instance/application*.

The MDE-Mega-Model and the important role of transformations and *transformation systems* can be explored in more detail in the paper by Favre et al. [9]. It also offers some insights on the correlation of (formal) languages and models, which enables a higher point of view on software in general. However, for this paper and its discussion of traceability the short introduction above is sufficient.

4 Traceability mega-models

So far we discussed traceability and mega-modeling separate from one another in a rather abstract manner. From here upon out we will combine both by studying several real-life approaches. However, the set-theoretic notion of the MDE-Mega-Model in §3.2 might have already revealed the connection between both topics.

4.1 MDE traceability

MDE needs formalization in order to handle traceability properly. But the definition of traces alone does not work well with automation. Therefor the concept of *traceability links* was introduced. Aizenbud-Reshef et al. define such links as *"[...] any relationship that exists between artifacts involved in the software-engineering life cycle. [...]"* [14]. Regarding models we have seen those relations in action:

$$aRb \text{ or } (a, b) \in R$$

This is the formalization utilized by the MDE-Mega-Model. We call the latter tuple representation a *relation instance*. Such instances denote a mapping or *link* between model-elements (modeled artifacts). So in fact the relations can be interpreted *traceability link models* and the corresponding instances as actual *traceability links*.

The MDE-Mega-Model defines only five relations, hence it is limited in inherent traceability. But this is sufficient due to its high level of abstraction and its objective to only model the fundamental relations to MDE. This set of relationships can easily be extended with additional traceability link models to achieve a higher degree of traceability.

4.2 MEGAF Architecture Frameworks

MEGAF[2] is an approach on global model management by Hillard et al. [4] utilizing mega-models to create reusable definitions of *architecture frameworks*.

Architecture frameworks as defined by the *ISO/IEC 42010 Software and System Engineering – Architecture Description* [15] standard are collections of coordinated viewpoints, conventions, principles and practices to create architecture descriptions for specific stakeholders. Where *architecture viewpoints* are collections of conventions, notations and modeling practices used to create concrete views which address domain specific system concerns for a distinct set of stakeholders. Their purpose is to capture and formalize a certain perspective.

According to the creators of MEGAF such architecture descriptions or frameworks alone have proven to be difficult to re-use and validate (consistency check). So in order to fix those issues, MEGAF shall provide features to:

- *store* architecture description elements (i.e. viewpoints, views, stakeholders, models, etc.), therefore making such elements reusable

[2] `http://megaf.di.univaq.it`

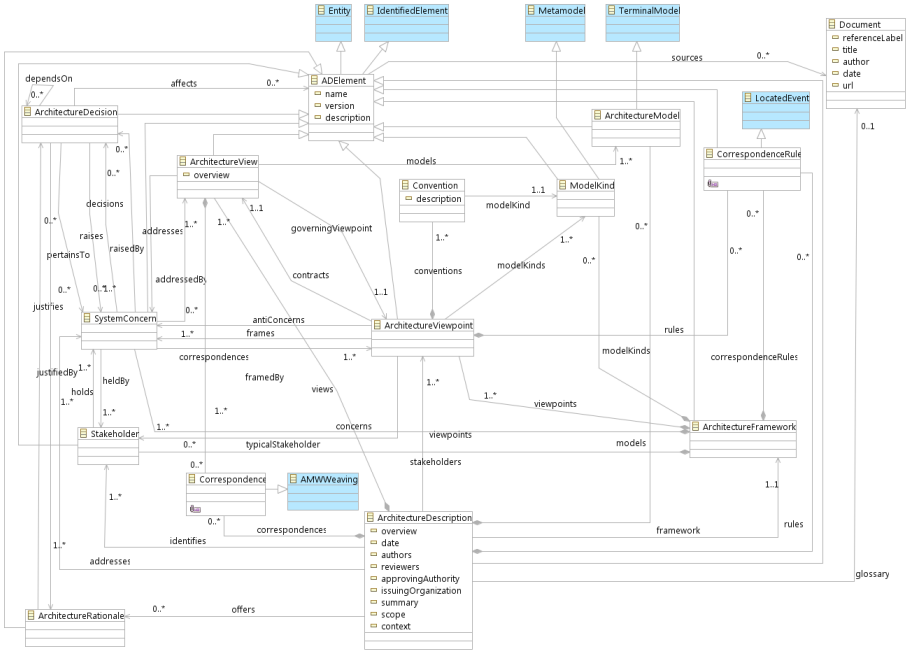

Fig. 5. The GMM4SA meta-model

- *define correspondence relations and correspondence rules* between architecture description elements
- *enable correctness and completeness checks* for architecture descriptions elements

To utilize mega-modeling techniques the assumption is made, that any architecture description element may conform to its meta-model. So a view may conform to a viewpoint, an architecture model may conform to a model kind, etc. This is straight forward and complies with the MDE world. The core of MEGAF is specified in *Global Model Management for Software Architecture* (GMM4SA[3]) meta-model shown in figure 5 as UML class diagram.

The diagram can be used to study the possible traceability of MEGAF. Because it is written in UML syntax, UML semantics apply and we already can identify several traceability links known from the MDE-Mega-Model. For example: `ArchitectureFramework` δ `ArchitectureViewpoint` or `ArchitectureViewpoint` ε `ADElement`. These relations exist inherent in class diagrams as association/aggregation/composition (δ) and generalization[4] (ε, χ). But this is trivial traceability information. The more beneficial traceability information here lies within

[3] `http://megaf.di.univaq.it/images/GMM4SA.png`

[4] UML generalization and realization do not map precisely to ε and χ . The meaning may vary with the employed perspective. The set theoretic perspective (ε) might be

the modeled context and intention. Because architecture description elements (children of `ADElement`) are formalizations of development process artifacts, all specified associations define a concrete traceability relation.

Lets consider the `ArchitectureViewpoint` class and its relations. In figure 5 we see on the right hand side that `ArchitectureViewpoint` δ `Convention` and `ArchitectureViewpoint` δ `ModelKind`. Those are hard facts and can be interpreted as *functional traces*. However on the left hand side wee see that an `ArchitectureViewpoint` *frames* a `SystemConcern`. In the object oriented programming way of understanding such diagrams we would say *frames* $\equiv \delta$. But if we also take the modeled context and intention of MEGAF into account, we discover how a `SystemConcern` traces to its holding `Stakeholder`, its pertaining `ArchitectureDecision` and the justifying `ArchitectureRationale`. Now we have information about *reason, context and decision* and a successful formalization of *non-functional traces*.

Wee see, the MEGAF meta-model supports traceability. Furthermore MEGAF and architecture frameworks in general target an very early phase of the development process. Using the higher order classification by Paige et. al [], MEGAF enables *pre-model* traceability. However we need to note that pre-model traceability is concerned with non-model- to early-model-artifact traceability. On the other hand MEGAF seems to heavily discourage the use of non-model-artifacts at all.

4.3 Dynamic hierarchical mega-models & Models at Runtime

4.4 Linguistic Architecture Analysis with MegaL

MegaL[3][5] is an approach on system understanding mainly concerned with the *linguistic architecture* of the system under study. It intents to uncover the systems technological footprint by revealing the used technological spaces and their interrelations. This is called a *linguistic* architecture because technological spaces of great interest are programming languages and their associated products.

In order to conduct a sufficient analysis MegaL needs to process the studied system against the

5 Conclusion

Conclusions are here.

References

1. A. Seibel, S. Neumann, and H. Giese. Dynamic hierarchical mega models: com- prehensive traceability and its efficient maintenance. Software & Systems Mod- eling, 9(4):493–528, 2010.

used for data modeling. The model-meta-model perspective (χ) might be used to describe inheritance or interface realization.

2. IEEE: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990
3. Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the Linguistic Architecture of Software Products.
4. R. Hilliard, I. Malavolta, H. Muccini, and P. Pelliccione. Realizing Architecture Frameworks Through Megamodelling Techniques. In Proc. of ASE'10, pages 305–308. ACM, 2010.
5. Ralf Lämmel and Andrei Varanovich. Interpretation of Linguistic Architectur. Unpublished, 2014
6. S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. Software and System Modeling, 9(4):529– 565, 2010.
7. Thomas Vogel, Andreas Seibel, and Holger Giese. The Role of Models and Megamodels at Runtime.
8. Lago, P., Muccini, H., van Vilet, H.: A scoped approacht to traceability management. J. Syst. Stoftw. 82(1), 168-182 (2009)
9. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. ENTCS, 127(3), 2004. Proc. of the SETra Workshop.
10. Pinheiro, F.A.C.: Requirements traceability. In: Sampaio do Prado Leite, J.C., Doorn, J.H. (eds.) Perspectives on Software Require- ments, pp. 93–113. Springer, Berlin (2003)
11. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the require- ments traceability problem. In: 1st IEEE International Require- ments Engineering Conference (RE'94) Proceedings, pp. 94–101. IEEE Computer Society, New York (1994)
12. Ramesh, B., Edwards, M.: Issues in the development of a require- ments traceability model. In: Proceedings of the IEEE Interna- tional Symposium on Requirements Engineering, pp. 256–259. IEEE Computer Society, New York (1993)
13. Simpson, J., Weiner, E. (eds.): Oxford English Dictionary, vol. 18, 2nd edn. Clarendon Press, Oxford (1989). ISBN 978-0-198- 61186-8
14. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. 45(3), 515–526 (2006)
15. ISO. ISO/IEC CD1 42010, Systems and software engineering — Architecture description (draft), January 2010.