A Comparison of Traceability in MDE Mega-Models

Maximilian Meffert

University of Koblenz-Landau

Abstract. Traceability and mega-modeling deliver a strong foundation to comprehend and work with big systems. But the research on traceability has not yet reached a common agreement nor has it reached standardization. Although especially MDE could benefit from a strong support of traceability for the engineering of new or the analysis of existing enterprise systems. This paper gives an overview of the current state of traceability and its support in selected MDE mega-models.

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

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 can be interpreted as the ability *"[...] to describe and follow the life of software artifacts [...]"* [8]. However, the research on traceability has not yet reached a common agreement nor has it reached standardization.

Other means to describe such big systems are mega-models, due to their high level of abstraction. Mega-models are commonly understood as models, which use other models as their elements. These models can be used to study technological spaces [7][10][6] or to provide the core for modeling tools [9][1], such as used in the Model Driven Engineering (MDE) world. Especially the latter might benefit from the support of traceability during the engineering of large enterprise systems.

In this paper we will outline the basics of traceability. That is, we will explore the several definitions of traceability given over time and introduce the nature of information traceability has to deal with. Further, we will elaborate on how traceability can be divided in different categories and explain what kind of objectives might be achieved with sufficient traceability support. Following that, we will deduce a comparison scheme to analyze several selected MDE mega-models regarding their support of traceability.

Contributions of this paper These are the contributions of this paper:

- We propose a simple, non-disjoint categorization of traceability objectives.
- We propose a simple comparison scheme to examine MDE mega-models regarding their support of traceability. That is, the comparison regarding traceability link types, traceability categories and traceability objectives.
- A comparison of the following mega-models: The MDE-Mega-Model [6], MEGAF [9], MegaL [10], Dynamic Hierarchical Mega-Models [1]

Non-Contributions of this paper These are non contributions of this paper:

- We do not propose a new definition for traceability.
- We do not propose a new understanding of traces and traceability links.
- We do not propose a new traceability classification.
- We do not propose new traceability objectives.

1.1 Road-map

§2 introduces the concept in terms of origins, motivation, terminology and classification of traceability. §3 introduces a comparison scheme and applies it to selected MDE mega-models. §4 concludes the paper.

2 Traceability

This section introduces the concept of traceability. It is based on Winkler and von Pilgrim [14], who compiled a complete survey on the subject. At first we give a brief overview of the historical origins of traceability. Secondly we explore several definitions of traceability given over time. Then we explain actual means to describe traceability information by introducing traces and traceability links. And finally we discuss the objectives of traceability.

2.1 Traceability Origins

Historically the research on traceability in software engineering originates from requirement engineering (RE) [14]. 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:

1. A raw vision is extended to concrete set of requirements,
2. then requirements are reworked into architecture,
3. and finally the architecture is transformed to an implementation.

Due to the incremental and iterative nature of such processes it seemed 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 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. Activities are seen as also well defined model-transformations. On top of that MDE is strongly interested in the automation of engineering processes.

2.2 Traceability Definitions

Traceability is subject of various research fields, hence there is no commonly agreed definition on what traceability actually is. In the following text we will discuss several definitions given over time.

Definition 1. *"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." [14]*

Traceability definition 1 is given by Lago et al. [8]. This definition is where the introductory citation is taken from. It 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. It could be the ability of an engineer, as traceability is also stated to be *"a means for modeling"*. But that would rather make it part of a personal skill set than an inherent aspect of software projects.

Definition 2. *"The ability to describe and follow the life of a requirement in both a forwards and backwards direction (i.e. from it origins, through its development and specification to its subsequent deployment and use, and through periods of on-going refinement in any of these phases)." [14]*

Traceability definition 2 is given by Gotel and Finkelstein [3]. It originates from RE research, as the subject of traceability is *"the live of a requirement"*. But it facilitates a broader view by exemplifying the life-cycle of requirements throughout a software project. That is by implicitly extending the definition to traceability for all artifacts, because most artifacts are deployed and used according to requirements [14].

Definition 3. *"[...] the ability to chronologically interrelate uniquely identifiable entities in an a way that matters. [...] [It] refers to the capability for tracing artifacts along a set of chained [manual or automated] operations." [14]*

Traceability definition 3 is given by Paige et al. [12]. It is expanded on all artifacts, which can exists during the a software development process. Also, it is stated that traceability creates a chronological order for the artifacts, following the applied operations. Winkler and von Pilgrim emphasized the fact, that these operations could either be manual or automated. For example a developer could have programmed a snipped of code himself, or he could have generated it. However, the definition is somewhat ambiguous, as artifacts are interrelated *"in a way that matters"*. This could be a weakness or a strength of this definition. In any case, it gives engineers the freedom to use traceability suitable to their needs.

Definition 4. *"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." [4].*

Traceability definition 4 is the first one given by the IEEE [4]. Here, traceability is not seen as an ambiguous ability. On the contrary, it is stated as a degree of historical or hierarchical relationships. The notion of degree expresses the hope, that traceability is a metric between software artifacts.

Definition 5. *"The degree to which each element in a software development products establishes its reason for existing." [4].*

Traceability definition 5 is the second one given by the IEEE [4]. It still emphasizes the fact, that traceability should be a measurable degree, not someones ability. Moreover in this definition product elements are made accountable for their own traceability. They are bound to justify their existence, which can include naming the relationships between them and their predecessors or masters, or vice versa.

This is a unsatisfactory starting position for further elaboration. So far, we presented five differing definitions, but there are more we have omitted. Some differences can be explained by their different research origins. However, those differences do not seem to be the relevant ones, as traceability of a requirement can easily extended to a general artifact traceability.

On the other hand, one difference which seems to matter is the one between the notions of ability and degree. The usage of the former is mostly ambiguous in terms of ownership. It is not clear if it is the ability of an engineer, a tool or of the artifact itself. The usage of the latter expresses the desire for traceability to be a metric, inherently there, only depending on measurement.

For this report, we do not propose a new definition. However, we encourage to understand traceability as the ability to make use of *traceability information* (traces). Therefore, it is not important who or what the carrier of the ability is. It can be there or not. But traceability information should exist in every development environment for every project. The nature of such information will be explored in the following sections.

2.3 Traceability Information

The second question, which motivated research on the subject of traceability was: On what information could one base requirement coverage? The vital idea to answer this question was the observation, any engineering activity leaves a trace. A very simple trace would be the outcome of a `diff` in a version control system, which shows the changes of an artifact between two revisions. However, traces can be more complex, if we consider the contents of the change. Source-Code could contain documenting comments, explaining what has changed and more importantly why. The following text will introduce the basic terminology and means for describing such traceability information.

Traces The most basic form of traceability information is called a *trace*. Such traces may appear in form of meta-information, describing hard facts like **what** happened, **who** did something or **when** it happened. I.e. most common file systems record the owner, modification date and the last access date of a file.

Opposing that, traces also may appear as *"... (non-) material indication or evidence ..."* [15] 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[1]. Such conversations are usually not transcribed, hence they are considered soft traceability information.

Following this separation in hard and soft fact traces, Pinheiro [13] 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"* [14] aspects. I.e. explaining correction due to edge case testing.

Traceability Links One IEEE definition of a trace is: *"[...] 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."* [4]. The notion of relationships describing or supporting traceability leads to the concept of *traceability links*. These links can be seen as a

[1] 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 comments or commit-messages.

special case of traces. They are stated to be instances of n-ary, multi-directional relations. However, the direction of a traceability link does not limit the navigation direction. Links can always be navigated in both forward and backward direction.

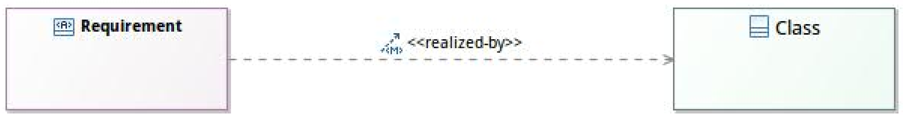


Fig. 1. Requirement realization link

The IEEE example for traceability links is shown in Figure 1, where the unidirectional realization link of a requirement by a class is denoted. Although, unlike traces, traceability links are not explicitly named by the IEEE Glossary. Therefore they are often used as synonyms for one another [14]. Figure 1 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. The relation between two versions of an artifact in the revision history or the Parent-Child relation between classes would be other examples of such traces.

Traceability Link Types Many researchers suggested that traceability links should be classified in types, depending on the relationship they are depicting [14]. Given the example shown in Figure 1, a suitable type for this link could be *Requirement Realization*. But, this would be a type for only one case. The suggested types usually share a higher granularity.

Aizenbud-Reshef et al.[2] included a simple classification in their definition for traceability links. Although the proposed class were meant to be actual types, they can be read as such. It contains: explicit links or as a result of transformations (code generation or reverse engineering), computed links based on existing information (code dependency analysis), statistically inferred links based on history by a revision systems [14]. Spanoudakis and Zisman [16] created a more complex type system from literature:

– **Dependency Links** denoting that changes in the depended artifact may result in changes in the depending artifact
– **Refinement Links** denoting abstraction or composition hierarchies between artifacts.
– **Evolution Links** denoting an Argument-Result relation between artifacts, the transformation of one artifact into an other
– **Satisfiability Links** denoting compliance of a lower artifact with a higher one
– **Overlap Links** denoting that artifacts share information, but differ in representation

– **Conflict Links** denoting that a there exists a conflict between artifacts
– **Rationalization Links** denoting the justification for the creation or evolution of artifacts
– **Contribution Links** denoting that a stakeholders shares interests with certain artifacts

Moreover, given the fact that traceability links can be seen as special cases of traces, the high order classification in Functional and Non-Functional could also be regarded as types.

2.4 Traceability Categorization

Similar to the several different definitions and type systems of traceability and traceability links, there is also no commonly agreed categorization system for traceability in general. However, most categorizations seem consider the phase or refinement nature of the traceability information.

In RE research, there is the distinction between *pre- and post- requirement specification traceability* by Gotel and Finkelstein [3]. These categories divide the development process in the two big phases before and after the specification of requirements. In contrast to that, Ramesh and Edwards [11] make a distinction between whether an artifact is traceable across refinement layers or not. This is called *horizontal and vertical traceability.* Figure 1 exemplifies vertical traceability, because realization is a relation between refinement layers. Horizontal traceability could exist alongside a class dependency (e.g. Dependency Injection).

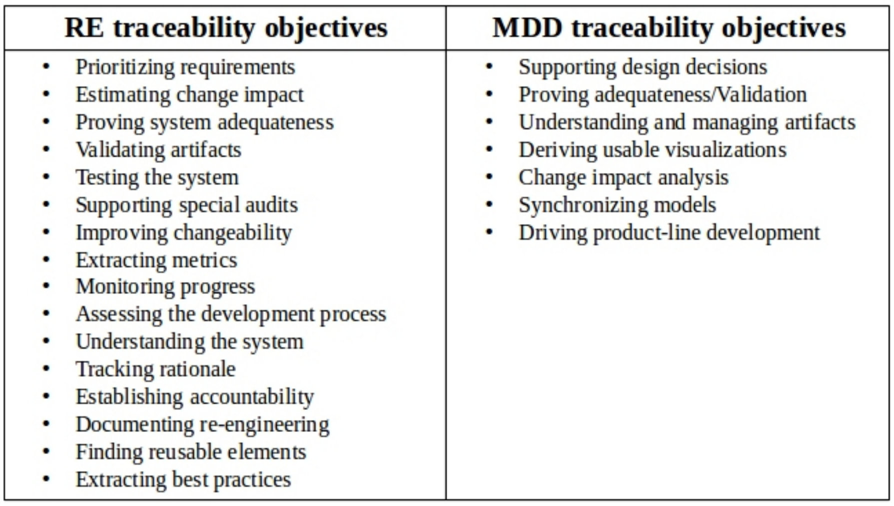
Additionally, there are also categorizations originating from MDE research. Paige et al. [12] also propose a big phase categorization system alongside the life-cycle of models:

– **Pre Model Traceability** considers traceability between the first non-model artifacts and the first models
– **Intra Model Traceability** considers traceability between models during model-refinement
– **Post Model Traceability** considers traceability between the final model and artifacts generated by that model

2.5 Traceability Objectives

Finally we want to further elaborate the benefits of traceability. The initial motivation behind the subject was the validation of requirement coverage. But general artifact or model traceability can achieve more than that.

Winkler and von Pilgrim [14] 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 MDE community. Additionally some objectives already correspond directly. For example *Estimating change impact* and *Change impact analysis* do not differ at all regarding



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

Fig. 2. Traceability Objectives identified by Winkler and von Pilgrim [14]

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, and cost. 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 to identify incompleteness (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, which 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.

All these objectives can be further categorized regarding stakeholders or general purpose. We propose the following categorization:

– **Technical Comprehension** considers objectives regarding the technical understanding of the system. The interests here lie in the supporting the actual engineering activities. (Supporting special audits, Improving changeability, Proving system adequateness, Extracting metrics, Tracking rationale,

Finding reusable elements, Documenting re-engineering, Supporting design decisions, Change impact analysis, Synchronizing models, Extracting best practices, Understanding the system, Extracting best practices)
- **System Management** considers objectives regarding management decisions. It is not concerned with technical details. The interests here lie in the monitoring of progress and accountability. Additionally, we are interested in the extraction of knowledge to support future decisions. (Prioritizing requirements, Proving system adequateness, Extracting metrics, Monitoring progress, Assessing the development process, Establishing accountability, Documenting re-engineering, Extracting best practices, Change impact analysis, Driving product-line development, Understanding the system)

However, those categories are not disjoint. That is because both fields do not exist in isolation. Management needs at least some general understanding of the system. On the other hand, technical decisions must also consider some management aspects. And eventually, a single achieved objective is beneficial for both fields. For instance, consider *Change impact analysis*: In the sense of Technical Comprehension, traceability can provide all entities, that also might need to be changed. In the sense of System Management, traceability can support estimating time and cost of the changes.

3 Comparison of MDE Mega-Models

One main goal of this paper is to provide a comparison of contemporary MDE Mega-Models regarding their support of traceability. 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 [10]. It can 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. So mega-models could actually describe mega-mega-models or even mega*-models.

Mega-models are valuable to MDE, because they can be used for large-scale modeling, i.e. models of big enterprise systems [6]. 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* [7][10][6]. However, the use of mega-models may not only be indicated by system size. Even minimal static (in a sense that content is not provided dynamically, e.g. by databases) 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.

Mega-models are also the hearts of MDE supporting tools. MDE tools provide functionalities to create and maintain digital representations of models, meta-models and model-transformations and the dependencies between them. Because over time the term traceability expanded from the tracing of requirements to general artifact/model traceability, we are interested to what extend it can be covered with such models. That is, what form of traceability information can be or is recorded by them.

In this section we first introduce the comparison scheme to compare the traceability aspects of the studied models. Then we will apply this scheme to the *MDE Mega-Model* by Favre et al. [6], *MEGAF* by Hillard et al. [9], *MegaL* by Lämmel and Varanovich [10] and finally *Dynamich Hierarchical Mega-Models* by Seibel et al. [1].

3.1 Comparison Scheme

In Section §2 we have introduced the basics of traceability. Because there is yet no definition that is commonly agreed upon, we do not use any definition given in §2.2 as distinguishing characteristic. However, we consider traceability information to be inherently existing. Therefore we investigate the nature of the traces, which are defined or can be excerpted. But we do not use the broad classification of Functional and Non-Function traces given by Pinhero et al.

Mega-models and models in general are used to interrelate their entities. Thus, following the leading notion of relationships, we consider traceability links as the main subjects of interest. We will use the types by Spanoudakis and Zisman introduced in §2.3 to describe the meaning of identified links.

Further we try to coordinate the models within the categories by Paige et al. introduces in §2.4. And finally we use our categorization of traceability objectives introduced in §2.5 to describe the general intention of the identified traceability.

3.2 The MDE-Mega-Model

The MDE-Mega-Model (Figure 3) by Favre et al. [6] is an approach, that aims to model typical MDE processes and relations. Although it is not yet complete, it already proves to be a powerful help for analyzing technological spaces. 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 usage of this term is not important, as everything can be seen as some kind of system [6]. 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 *entity of interest*.

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

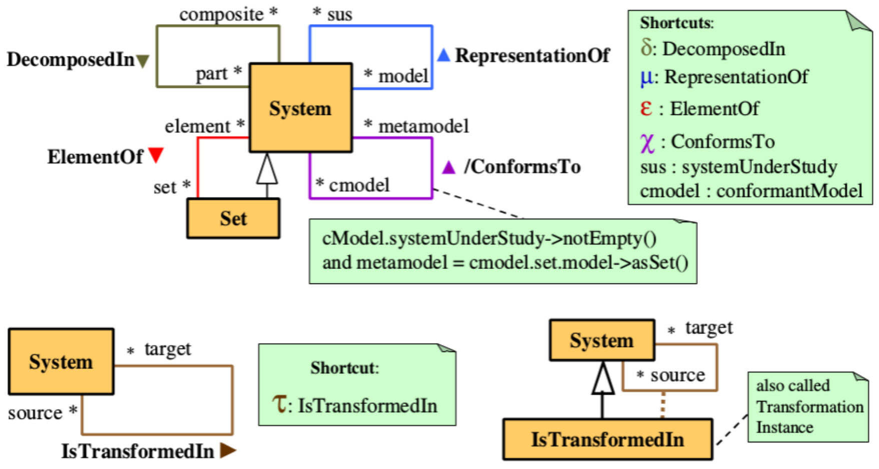


Fig. 3. The MDE-Mega-Model by Favre et al. [6]

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

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: 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)+` is 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 expressions 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:

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

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 by `(/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. [6]. 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.

Traceability Links There are five relations defined in the MDE-Mega-Model and all these links carry traceability information. The most simple ones are δ and ε, both are **Refinement Links**. Refinement Links denote hierarchies of abstraction or dependencies of aggregation and composition. The δ relationship exactly does the latter. It states that a composite can be broken down into a smaller part. In this regard, the ε shares that characteristic for aggregation. But it can also be interpreted as abstraction.

Other simple relations for comparison are χ and τ. χ defines conformance in the sense of meta-modeling. This is a **Satisfiability Link**, which denotes upstream compliance. However, χ is also a **Dependency Link**. That is because changes in a meta-model can result in necessary changes on downstream entities to preserve conformance. In that regard the traceability link types by Spanoudakis and Zisman are not exclusive. τ is an **Evolution Link**, which is defined to capture that one entity is transformed into another. This could be just simple content modification, where the evolution would correspond to versioning, or it could be the generation of a parser.

The last relationship of the MDE-Mega-Model is μ. This can be an **Overlap Link** in combination with τ, but this traceability link is not explicitly in the model. Consider two transformations `toXml` and `toJson` which transform an artifact to XML or JSON:

$$a\ \texttt{toXml}\ a_{xml},\ a\ \texttt{toJson}\ a_{json}$$

In the notion of the MDE-Mega-Model $a_{xml}\mu a$ and $a_{json}\mu a$ would hold, so a_{xml} and a_{json} would be connected via an Overlap Link.

Traceability Categories The MDE-Mega-Model is a theoretical approach to study MDE as a whole. Hence, it cannot be clearly be categorized in one of the three categories **Pre Model**, **Intra Model** or **Post Model**. It rather can be used in all of these phases of the MDE process.

Traceability Objectives As stated before, the MDE-Mega-Model as a theoretical approach on MDE. Its primary use case is academic research on MDE relations and dependencies [6]. Thus major objective category that can be targeted with this model is **Technical Comprehension**. However, since both categories are not fully disjoint, it is also possible to achieve objectives in the System Management category. We only consider the intended use.

3.3 MEGAF

MEGAF[2] is an approach on global model management by Hillard et al. [9] utilizing mega-models to create reusable definitions of architecture frameworks.

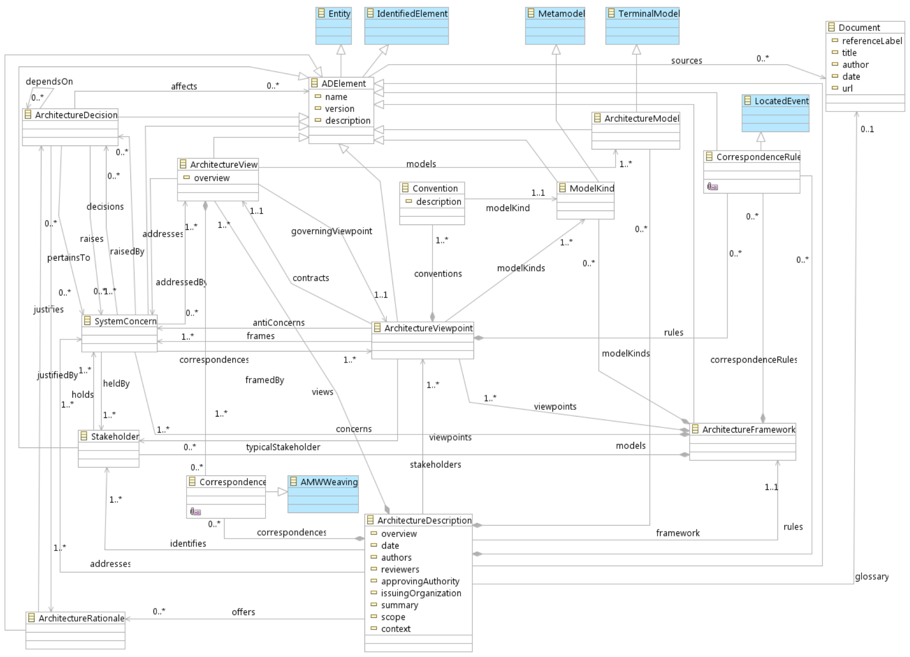


Fig. 4. The GMM4SA meta-model

Architecture frameworks are defined by the *ISO/IEC 42010 Software and System Engineering – Architecture Description* Standard [5]. They are collections of coordinated viewpoints, conventions, principles and practices to create

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

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) [9]. 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
- *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 4 as UML class diagram.

Traceability Links 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 (ε). Therefore, and because of the size of the GMM4SA, there are many **Refinement Links** that can be found in this model.

The relation `ArchitectureViewpoints contracts ArchitectureView` denotes that views have conform to a viewpoint. So both entities are connected via an **Satisfiability Link**, and likewise to the MDE-Mega-Model satisfiability creates a **Dependency Link** in order to preserve conformance.

But in contrast to the MDE-Mega-Model the GMM4SA contains `Stakeholders` and an `ArchitectureRationale`, which enables additional traceability capabilities. For instance the relationship `Stakeholder holds SystemConcern` creates a **Contribution Link**. And the relation `ArchitectureRationale justifies ArchitectureDecision` records extra background information for the development process. Thus it creates a **Rationalization Link**, which comparable to satisfiability propagates the creation of a **Dependency Links**. A change in the rationale could very likely result in downstream changes.

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

Traceability Categories MEGAF aims to create tool support for reusable architecture frameworks. Architecture frameworks can be used to inspect the model at any time during development process from a higher perspective. So, using the categorization by Paige et. al [12], MEGAF enables **Pre Model**, **Intra Model** and **Post Model** traceability.

Traceability Objectives MEGAF is an approach on global model management. It aims to give a high level overview over a system, and records important traceability information like contribution (stakeholders) and rationalization (architecture rationale, system concerns). Both aspects are valuable to Technical Comprehension and System Management. However, the high abstraction of architecture frameworks is more suitable for **System Management** objectives than it is for Technical Comprehension.

3.4 MegaL

MegaL is an approach on system understanding mainly concerned with the linguistic architecture of the system under study [7][10]. It intents to uncover its 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.

```
Set < Entity // Sets such as languages; see below
Language < Set // Languages as sets, e.g., sets of strings
Technology < Entity // Technologies in the sense of conceptual entities
Artifact < Entity // Artifacts as entities with a physical manifestation
File < Artifact // Files as a common kind of artifact
Function < Set // A function such as the meaning of a program
FunctionApplication < Entity // A particular application of a function

elementOf < Entity * Set // Membership in the set−theoretic sense
conformsTo < Artifact * Artifact // Conformance in the sense of metamodeling
defines < Artifact * Entity // Such as a grammar defining a language
domainOf < Set * Function // The domain of a function
rangeOf < Set * Function // The range of a function
inputOf < Entity * FunctionApplication // The input of a function application
outputOf < Entity * FunctionApplication // The output of a function application
partOf < Entity * Entity // A physical or conceptual containment relationship
```

Fig. 5. Definition of MegaL Entities and Relationships [10]

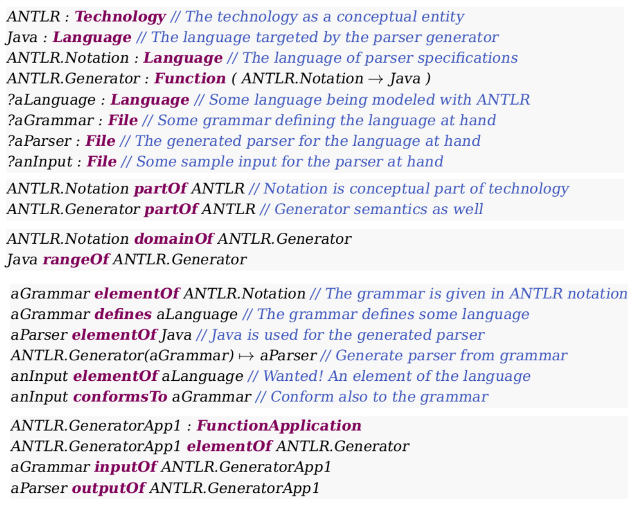
In fact, MegaL is not only a mega-model, it is rather a mega-model description language. Figure 5 shows the definition of the main entities and relationships of MegaL. We see, it is closely related to the MDE-Mega-Model, as all relations defined there make an occurrence here too. However, the relations of MegaL have stricter semantics. Each relation is defined as the Cartesian product of concrete entities.

As software language, MegaL is designed to be interpreted. Given a mega-model defined in MegaL, this mega-model can be applied to any system [10]. To do so, Lämmel an Varanovich designed a extendable interpreter, which loads a mega-model written in MegaL and a configuration file, which contains an URI to the resource to analyze and a specification of suitable resolvers and evaluators. Where resolvers are programs, which resolve the resource URI following Linked-Data principles (for instance a resource may be located at a distant server). And evaluators are programs, suitable to evaluate the various MegaL relationships (e.g. the element-of-Java relation) [10].

Unlike the models before, MegaL has build-in capabilities for *traceability recovery* [10]. Evaluators can have the ability to record traceability links as pairs of URIs:

⟨ "http://.../MegaLParser.java/class/MegaLParser/method/megamodel/1" ,
"http://.../MegaL.g4/grammar/megal/rule/megamodel/1" ⟩

Here, linking the `conformsTo` relationship of class-method in the java file MegaL-Parser.java to the grammar rule found in the file MegaL.g4 [10].



```
ANTLR : Technology // The technology as a conceptual entity
Java : Language // The language targeted by the parser generator
ANTLR.Notation : Language // The language of parser specifications
ANTLR.Generator : Function ( ANTLR.Notation → Java )
?aLanguage : Language // Some language being modeled with ANTLR
?aGrammar : File // Some grammar defining the language at hand
?aParser : File // The generated parser for the language at hand
?anInput : File // Some sample input for the parser at hand

ANTLR.Notation partOf ANTLR // Notation is conceptual part of technology
ANTLR.Generator partOf ANTLR // Generator semantics as well

ANTLR.Notation domainOf ANTLR.Generator
Java rangeOf ANTLR.Generator

aGrammar elementOf ANTLR.Notation // The grammar is given in ANTLR notation
aGrammar defines aLanguage // The grammar defines some language
aParser elementOf Java // Java is used for the generated parser
ANTLR.Generator(aGrammar) ↦ aParser // Generate parser from grammar
anInput elementOf aLanguage // Wanted! An element of the language
anInput conformsTo aGrammar // Conform also to the grammar

ANTLR.GeneratorApp1 : FunctionApplication
ANTLR.GeneratorApp1 elementOf ANTLR.Generator
aGrammar inputOf ANTLR.GeneratorApp1
aParser outputOf ANTLR.GeneratorApp1
```

Fig. 6. MegaL Example: ANTLR Mega-Model[10]

Figure 6 depicts the example usage of MegaL. It shows the specification of a mega-model for the ANTLR parser generator. This example includes some shorthand notation:

– `ANTLR.Notation` (dot notation) is dissolved as part-of relationship shown in the second paragraph of Figure 6
– `ANTLR.Notation` $\rightarrow$ `Java` is dissolved as function definition in form domain-of and range-or relationships shown in the third paragraph of Figure 6
– `ANTLR.Generator(aGrammar)` $\mapsto$ `aParser` is dissolved as function application shown in the last paragraph of Figure 6

Traceability Links As noted before, MegaL already has the capabilities to capture traceability links. However, the defined captured relationships are recorded as is. For instance the `conformsTo` relation is recored as pair of URIs.

We apply the type system of Spanoudakis and Zisman in order to compare MegaL to the other presented models. Although, because of the great analogy to the MDE-Mega-Model by Favre et al. many traceability link types can be adopted. That is for the relationships `elementOf`, `partOf` and `conformsTo` and `FunctionApplication` entities. Therefore the types **Refinement Link**, **Satisfiability Link**, **Dependency Link** and **Evolution Link** including the possible **Overlap Link** exist here for the same reasons.

However, transformation here are modeled in more detail. Functions are defined via the `domainOf` and `rangeOf`. Thus there are also refinement links between the modeled function and other entities. The same holds for the `inputOf` and `outputOf` relations of function applications.

Traceability Categories MegaL is intended to analyze existing systems and technological spaces. In that regard it can be categorized to support **Post Model** traceability. However, like its related model, the MDE-Mega-Model, it is also meant for research purposes, With its flexible design regarding resolver and evaluator plug-in-system, it could easily be used to analyze models from the pre and intra model phases of MDE processes.

Traceability Objectives Lämmel and Varanovich clearly state that MegaL is especially designed for *"[...] modeling the linguistic architecture of software systems, i.e., a system's architecture in terms of relationships between conceptual entities such as languages and technologies as well as actual entities [...]"* [10]. This is best out with the objective *Understanding the system.* Hoever, this objective is found in both categories. But the extend, to which **Technical Comprehension** is offered by MegaL seems to exceed the level understanding needed for System Management by far.

3.5 Dynamic Hierarchical Mega-Models

Dynamic Hierarchical Mega-Models (DHMM) are an approach on traceability link management by Seibel et al. [1]. It is meant to be used as the core of a proposed model environment. The DHMM actually models how traceability links interrelate with entities and also models their nature regarding types. In that aspect it differs greatly from the previously analyzed models.

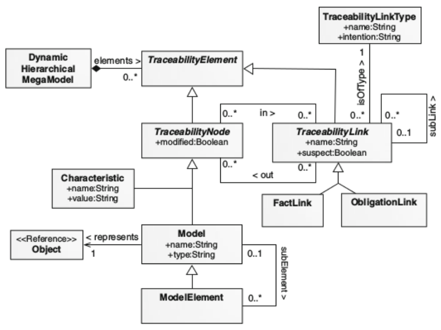


Fig. 7. Meta-Model of the Dynamic Hierarchical Mega-Model [1]

Figure 7 depicts the relevant part of the DHMM meta-model. It shows that a traceability link connects two traceability nodes like models or model elements. It is also shown that a traceability link is of exactly one type and either is a FactLink or an ObligationLink. Fact links describe simply that a modeled intention between to artifacts hold, thus they are facts [1]. Obligation links describe that *"the instantiation of some arbitrary operation [...] satisfy the intention [and thus the obligation] should hold"* [1]. These links a meant to be placed by human modelers. In that regard Seibel et al. seem to follow the classification of traces in Functional (fact) and Non-Functional (obligation) by Pinheiro et al. [13]. Where the former trace have to obey some sort of formalism and the latter need to cover *"reason, context, decision, and technical"*[13].

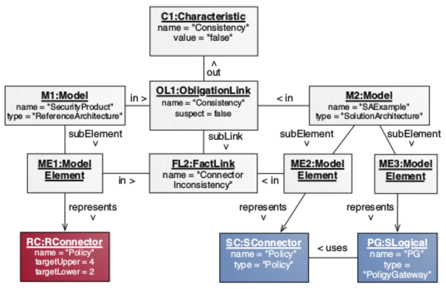


Fig. 8. Top-Down Hierarchy of Traceability Links [1]

The key aspect of DHMMs is the notion hierarchy. In Figure 7 it is defined by the sub-link self-association. Any traceability link can have multiple sub-links. This is exemplified in Figure 8, which shows a top-down hierarchy. A modeler has placed an obligation link between to models to ensure the consistency between them. Figure 8 further shows, that a validation mechanism has found an inconsistency and therefore deduced a fact link between the model elements one hierarchy level beneath. The inconsistency here lies in the fact that the RConnector policy demands at least two connections. However, there is only one SConnector. DHMMs can also have bottom-up hierarchies, but we omit those here.

Traceability Links Because the DHMM is an actual traceability link management model, which includes the model for link types, we cannot identify any further types following the comparison scheme. So, we could just conclude that custom types are allowed. However, Figuer 8 shows the deduction of a traceability link denoting an inconsistency. This can be classified as a **Conflict Link** according to Spanoudakis and Zisman. Moreover, one big aspect of DHMMs is the validation of created traceability links, which can easily result in many of such Conflict Links.

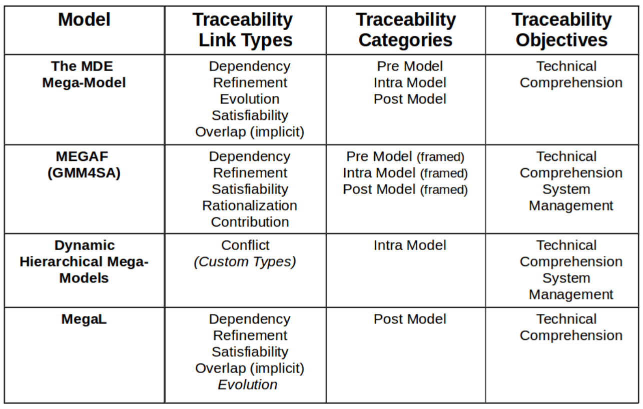
Traceability Categories The DHMM is the heart of a modeling environment proposed by Seibel et al [1]. Therefore, it is meant to create and maintain models. That makes it fall in the **Intra Model** traceability category. However, depending on the extend of the model managed by the DHMM, in the sense that artifacts generated by such model are also managed, it also falls in the Post Model traceability category.

Traceability Objectives Similar to traceability categories, DHMMs are meant for tool support grade of traceability. Traceability link types can be dynamically defined by modelers [1]. Any kind of traceability information could be recorded and analyzed to provide knowledge for System Management. However, Seibel et al. made great efforts to optimize DHMMs for validation and automation [1]. The intended use is clearly stated to be for human modelers, i.e. technical staff. So it seems the main objectives of DHMMs lie in the category of **Technical Comprehension**.

4 Conclusion

We compared the MDE-Mega-Model, MEGAF, MegaL and Dynamic Hierarchical Mega-Models regarding their support of traceability. We used the traceability types proposed by Spanoudakis and Zisman [16], the traceability categories by Paige et al. [12] and our own categorization of the traceability objectives collected by Winkler and von Pilgrim [14] as characteristics of our comparison scheme.

Figure 9 shows the result of our comparison. It shows that the rather academical approaches, MegaL and the MDE-Mega-Model, are more interested in the technical aspectes of MDE. Therefore their main traceability objective seems to be **Technical Comprehension** and only structural traceability links can be found (**Refinement, Dependency, Evolution, Overlap and Satisfiability**). In contrast to that, the approaches with enterprise tool support in mind,



Model	Traceability Link Types	Traceability Categories	Traceability Objectives
The MDE Mega-Model	Dependency Refinement Evolution Satisfiability Overlap (implicit)	Pre Model Intra Model Post Model	Technical Comprehension
MEGAF (GMM4SA)	Dependency Refinement Satisfiability Rationalization Contribution	Pre Model (framed) Intra Model (framed) Post Model (framed)	Technical Comprehension System Management
Dynamic Hierarchical Mega-Models	Conflict *(Custom Types)*	Intra Model	Technical Comprehension System Management
MegaL	Dependency Refinement Satisfiability Overlap (implicit) *Evolution*	Post Model	Technical Comprehension

Fig. 9. Comparison Result of selected MDE Mega-Models

MEGAF and DHMM, enable traceability, which can provide additional information (**Contribution, Rationalization and Conflict**). Moreover, they seem to also aim to support **System Management** as traceability objective.

The traceability categories according to Paige et al. seem to be a rather poor characteristic for comparison for mega-models. But that is not because of the categories itself than it is because of the varying abstraction levels of different mega-models. For instance architecture frameworks formalize perspective. Such perspective could be applied throughout the all categories for all model granularities. That is why we annotated the categories in the result matrix with *framed*. Event the theoretical approaches (MegaL) could be extended to be used throughout the whole development process, although this is not always intended.

Future work on this topic could include:

- the work on a more suitable categorization for traceability in general
- the study of traceability links, which exist only implicitly, such as the overlap link in the MDE-Mega-Model and MegaL.

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.
2. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Syst. J. 45(3), 515–526 (2006)
3. 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)
4. IEEE: IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990

5. ISO. ISO/IEC CD1 42010, Systems and software engineering — Architecture description (draft), January 2010.
6. J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution through Transformations. ENTCS, 127(3), 2004. Proc. of the SETra Workshop.
7. Jean-Marie Favre, Ralf Lämmel, and Andrei Varanovich. Modeling the Linguistic Architecture of Software Products.
8. Lago, P., Muccini, H., van Vilet, H.: A scoped approacht to traceability management. J. Syst. Stoftw. 82(1), 168-182 (2009)
9. 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.
10. Ralf Lämmel and Andrei Varanovich. Interpretation of Linguistic Architectur. Unpublished, 2014
11. 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)
12. Paige, R.F., Olsen, G.K., Kolovos, D.S., Zschaler, S., Power, C.: Building model-driven engineering traceability classifications. In: ECMDA Traceability Workshop (ECMDA-TW) 2008 Proceed- ings, pp. 49–58. Sintef, Trondheim (2008). ISBN 978-82-14- 04396-9
13. 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)
14. 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.
15. Simpson, J., Weiner, E. (eds.): Oxford English Dictionary, vol. 18, 2nd edn. Clarendon Press, Oxford (1989). ISBN 978-0-198- 61186-8
16. Spanoudakis, G., Zisman, A. Software traceability: a roadmap. In: Chang, S.K. (ed.) Handbook of Software Engineering and Knowl- edge Engineering, vol. 3—Recent Advances, pp. 395–428. World Scientific, Singapore (2005). ISBN 978-9-8125-6273-9
17. Thomas Vogel, Andreas Seibel, and Holger Giese. The Role of Models and Megamodels at Runtime.