Managing Artifacts with a Viewpoint-Realization Level Matrix

Jochen M. Küster, Hagen Völzer, Olaf Zimmermann

IBM Research - Zurich, Säumerstr. 4, 8803 Rüschlikon, Switzerland
{jku,hvo,olz}@zurich.ibm.com

Abstract. We propose an approach to artifact management in software engineering that uses an *artifact matrix* to structure the artifact space of a project along stakeholder viewpoints and realization levels. This matrix structure provides a basis on top of which relationships between artifacts can be defined, such as consistency constraints, tracebility links and model transformations. The management of all project artifacts and their relationships supports collaboration across different roles in the development process as well as change management and agile practices. Our approach is highly configurable to facilitate adaptation to different development methods and processes. It provides a basis to develop and/or to integrate generic tools that can flexibly support such different methods. In particular, it can be leveraged to improve the transition from requirements analysis to architecture design.

1 Introduction

The state of the art in requirements engineering and software architecture has advanced significantly in recent years. Mature requirements and software engineering methods such as the Unified Process (UP) [18] specify processes to be followed and artifacts to be created in application development and integration projects. In requirements engineering, informal and formal notations as well as modeling techniques are available to the domain analyst, for example vision statements, strategy maps, business process models, user stories, and use cases [30]. In software architecture, both lightweight and full-fledged architecture design processes exist. Architectural tactics, patterns, and styles help architects to create designs from previously gained experience that is captured in reusable assets [2, 6]. Techniques and tools for architectural decision capturing and sharing have become available [19, 32].

When applying these or other state-of-the-art methods, techniques and tools on large projects, requirements engineers and architects often produce a large amount of rather diverse artifacts. If this is the case, it is challenging to maintain the consistency of these artifacts and to promote their reuse. Likewise, the large body of existing work makes it difficult for creators of requirements engineering and architecture design methods to assemble domain-specific methods that are both comprehensive and consumable; Service-Oriented Architecture (SOA) design exemplifies this problem [32]. The same issue makes it hard to build tools that flexibly support such methods for different domains.

Relationships between artifacts do not only have to be understood both *within* requirements analysis and *within* architecture design; as motivated in previous chapters of this book, a seamless transition *between* these analysis and design activities is particularly important. For instance, the dependency between architecturally significant requirements and the architectural decisions that are required to satisfy these requirements often remains undocumented. This is unfortunate because the requirements engineer, who is familiar with the application domain, can advise the architect on domain-specific design issues that arise from common requirements in the domain. Examples of such requirements are industry-specific process models and regulatory compliance rules. The architect, on the other hand, has tacit knowledge that makes him/her pre-select certain patterns, technology standards and implementation assets. The traceability links to the requirements that are satisfied by these assets are often not made explicit; therefore it is difficult to evaluate whether a given design is an adequate solution for a particular business domain and problem. According to our industry project experience, this gap between requirements engineering and architecture design delays projects or leads to project failure; application maintenance may also suffer.

Similar problems can be observed between other roles and viewpoints. For instance, test cases should be derived from functional and non-functional requirements; they should also examine design hot spots such as single points of failure (if high availability is a desired quality attribute) and scalability bottlenecks (if many concurrent users are likely to perform complex operations concurrently). Hence, the artifacts created by testers should be aligned with those used by requirements engineers, architects, and developers. At a minimum, terminologies should be aligned, traceability links be defined, and continuous refinement and change of artifacts be supported across these four roles.

The outlined problems in managing requirements engineering, architecture design, and other software engineering artifacts can be abstracted and generalized:

1. *Method definition*: When a software engineering method is defined, either by creating a new method from scratch for a new domain or by instantiating, tailoring, and combining existing methods, one has to define which artifacts of which type to include and which of their dependencies to trace. Specifically for requirements engineering and architecture design, one has to determine what the deliverables of specific analysis or design activities are and how they relate to each other (e.g., use cases and class diagrams defined in the Unified Modeling Language (UML) [28]).
2. *Tool design:* It should be defined which tools, if any, are used to create and manage the artifacts defined in a method. These tools should support the flexible application of a method beyond the provision of simple editors for artifact notations such as UML. Specifically for requirements engineering and architecture design, it should be defined how the requirements engineering and the architecture design tools are organized and how they interface with each other. E.g., are the same tools used by both roles? If so, are different UML profiles used? How can the required architectural decisions be identified in requirements artifacts?

To address these two general problems, we propose an integrated, model-driven artifact management approach. This general approach can be leveraged to specifically improve the transition from requirements engineering to architecture design. In the center

of our approach is the *Artifact and Model Transformation (AMT) Matrix*, which provides a structure for organizing and maintaining artifacts and their dependencies. The default incarnation of our AMT Matrix is based on stakeholder *viewpoints* and analysis/design refinement levels, which we call *realization levels*, two common denominators of contemporary analysis and design methods. Relative to the two general artifact management problems, our AMT approach contributes the following:

1. *Method definition*: With the AMT matrix, our approach provides a generic structure to support the definition of methods. It contributes a metamodel that formalizes the AMT Matrix and its relationship to software engineering artifacts.
2. *Tool design:* With the AMT matrix, our approach provides a foundational structure for designing and integrating tools that provide transformations between different artifacts, create traceability links automatically, validate consistency, and support cross-role and cross-viewpoint collaboration.

Via instantiation and specialization, these two general contributions can be leveraged to solve our original concrete problem of better aligning and linking requirements engineering and architecture design artifacts (e.g., user stories, use cases, logical components, and architectural decisions).

The remainder of the chapter is structured in the following way. In the next section, we clarify fundamental literature terms such as viewpoint and realization level and introduce our general concepts on an informal level. After that, we formalize these concepts in a metamodel. The concepts and their formalization allow us to define cross-viewpoint transformations and traceability links between requirements engineering and architecture design artifacts. These solution building blocks form the core contribution of this chapter. In the remaining sections of the chapter, we provide an example how our concepts can be applied in practice, outline the implementation of a tool prototype and discuss related work. We conclude with summary and a discussion of open issues.

2 The Artifact and Model Transformation (AMT) Matrix

Both in requirements engineering and in software architecture design, model-driven software development is applied in various forms. Maturity levels and practitioner adoption vary by domain. For instance, software engineering processes such as Object-Oriented Analysis and Design (OOAD) [4] and modeling languages such as the Unified Modeling Language (UML) [28] are successfully adopted in embedded systems engineering and enterprise application development today. Well-crafted models facilitate communication between stakeholders; formal models can be processed by tools in support of automation. A key concept of model-driven software development is to construct models that describe the system under construction from different viewpoints and at different levels of abstraction; these models then become the artifacts to be created when following a particular method. The information found in already existing models serves as input to create model artifacts. For example, in OOAD and Component-Based Development (CBD), requirements analysis artifacts such as UML use cases may serve as input to the construction of architecture design artifacts such as functional component models expressed as UML class diagrams [8].

To overcome the artifact management problems identified in the previous section, we structure the model space of a project as an Artifact and Model Transformation (AMT) matrix. The goal of the AMT matrix is to organize the model space according to the concepts in the chosen software engineering method. The default incarnation of our AMT matrix has two dimensions: The horizontal dimension represents disjoint/discrete stakeholder *viewpoints* as defined in the IEEE 42010 specification for architecture descriptions [24]; the vertical dimension of the matrix represents *realization levels* as defined in methods promoting an incremental and iterative refinement of artifacts.

For instance, the architecting process defined by Eeles and Cripps [8] distinguishes stakeholder-specific, role-based viewpoints such as 'requirements', 'functional', and 'deployment'; their two realization levels are the platform-independent 'logical level' and the platform-specific 'physical level'.[1] The AMT matrix entry for functional design on the logical refinement level may then list, for example, UML class diagrams and sequence diagrams as the artifact types that populate this matrix entry.

Both AMT matrix dimensions can be configured for a particular software engineering method via the viewpoints and realization levels defined in the method. An AMT matrix for UP, for instance, differs from an AMT matrix for an agile process in terms of number, names, and semantics of viewpoints and realization levels. UP leverages Kruchten's original 4+1 viewpoint model; these viewpoints become the columns of the matrix. Elaborating a design via multiple iterations requires touching already existing artifacts multiple times; by defining one realization level row for each iteration, these different stages of the artifact evolution can be distinguished from each other.

AMT matrix entries can be connected by *traceability links* and *transformations* between artifacts and individual artifact elements (e.g., between steps in a use case model and operations/methods in a UML class diagram). To enforce its design practices, the software engineering method determines which links and transformations are valid. For example, it might not permit to bypass a realization level or to increase the realization level and switch the viewpoint in a single atomic transformation.

In the following, we introduce our AMT matrix management approach in three steps:

1. Specify AMT matrix dimensions (i.e., viewpoints and realization levels by default).
2. Position method-specific artifact types in AMT matrix entries (according to their purpose).
3. Populate a project-specific AMT matrix instance with artifacts (according to their type).

Step 1: Specify AMT matrix dimensions. Our first step is preparatory. As outlined above, we propose to organize all types of models and other artifacts defined in a method (and, later on, models and other artifacts created on projects) in a multi-dimensional matrix structure. In this preparatory step, we define how many dimensions are used, what the semantics of these dimensions are, and how these dimensions are structured and sourced.

[1] Note that Eeles and Cripps [8] use the terms 'logical' and 'physical' for realization levels whereas the 4+1 viewpoint model in UP uses them for particular viewpoints.

The default incarnation of an AMT matrix has two dimensions. We decided to combine two problem solving strategies that are promoted by many contemporary software engineering methods and commonly used in many other engineering disciplines:

- Partitioning by stakeholder-specific *viewpoints* [18, 24].
- Incremental refinement via *realization levels* [8, 15].

The two-dimensional default AMT matrix structure resulting from these considerations is shown in Figure 1. In the remainder of this chapter, we work with this default structure; adding dimensions is subject to future research.

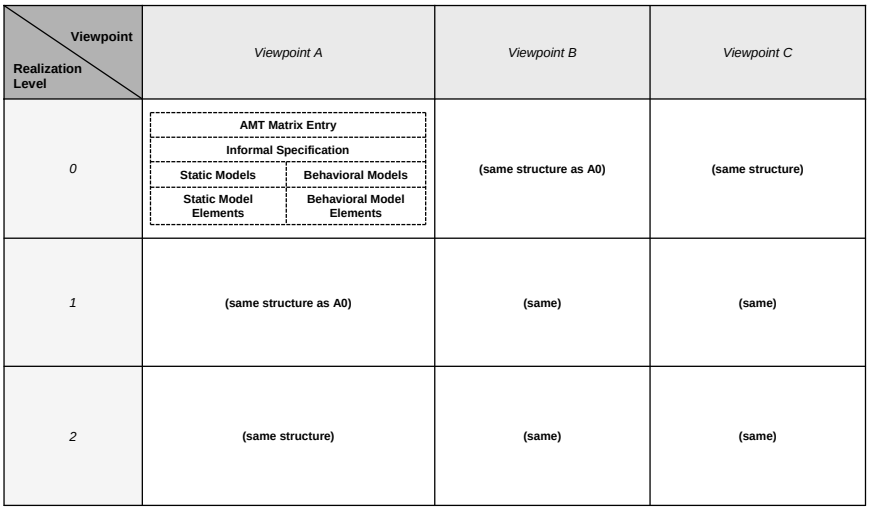

Fig. 1. Artifact and Model Transformation (AMT) matrix structure with entry content

Each entry in the matrix serves one particular, well-defined analysis or design purpose. In the horizontal dimension, the stakeholder viewpoints differ in terms of analysis/design concerns addressed as well as notations used and education/experience required to create artifacts. In the vertical dimension, each level has a certain depth associated to it such as platform-independent conceptual modeling, technology platform-specific but not yet vendor-specific modeling, and executable/installable platform-specific modeling and code. Both informal and formal specifications may be present in each matrix entry; both static structure and dynamic behavior of the system under construction are covered.

Our rationale for making discrete viewpoints a default matrix dimension is the following: Each such viewpoint takes the perspective of a single stakeholder with a par-

ticular concern. This makes the artifacts of a viewpoint, i.e., diagrams and models, consumable as it hides unnecessary details without sacrificing end-to-end consistency.[2]

We decided for realization levels as our second dimension because they allow elaborating analysis results and design artifacts in an iterative and incremental fashion without losing the information from early iterations when refining the artifacts. This is different from versioning a single artifact to keep track of editorial changes, i.e., its evolution in time (in the absence of dedicated artifact management concepts such as those presented in this chapter, the current state of the art is to define naming conventions and use document/file/model versioning to manage artifacts and organize the model space in a project). The same notation can be used when switching from one realization level to another, but more details be added. For instance, a UML class diagram on a logical refinement level might model conceptual patterns and therefore not specify as many UML classes and associations as as Java class diagram residing on the physical realization level. Furthermore, different sets of stereotypes might be used on the two respective levels although both take a functional design viewpoint.

Realization levels support an iterative and incremental work organization which helps to manage risk by avoiding common pitfalls such as big design upfront (a.k.a. analysis paralysis or waterfall) but also the other extreme, ad hoc modeling and developer anarchy[3]. Instances of this concept can be found in many places. For instance, database design evolves from the conceptual to the logical to the physical level. Moreover, the Catalysis approach and Fowler in UML Distilled [14] promote similar approaches for UML modeling (from analysis to specification to implementation models). To give a third example, an IBM course on architectural thinking recommends the same three-step refinement for the IT infrastructure (deployment) viewpoint dealing with data center locations, hardware nodes, software images, and network equipment. Finally, the distinction between platform-independent and platform-specific models in Model-Driven Architecture can be seen as an instance of the general approach of refinement levels as well. Additional rationale for selecting viewpoints and realization levels as primary structuring means can be found in the literature [8, 32].

Step 2: Position method-specific artifact types in AMT matrix entries. Our second step is performed by method creators and tool engineers. Each method and each analysis or design tool supporting such method is envisioned to populate the AMT matrix structure from step 1 with artifact types for combinations of viewpoints and realization levels. It is not required to fully populate the matrix in this step; it rather serves as a structuring means. However, gaps should not be introduced accidentally; they should rather result from conscious engineering decisions made by the method creator or tool engineer.

To give an example, we now combine agile practices, OOAD, and CBD; our concepts are designed to work equally well for other methods and notations. Figure 2 shows an exemplary AMT matrix; its viewpoints stem from three references [8, 18, 31] and the

[2] Cross-cutting viewpoints such as security and performance have different characteristics; as they typically work with multiple artifacts, they are less suited to serve as matrix dimensions. However, such viewpoints can be represented a slices (projections) through an AMT matrix, e.g., with the help of keyword tags that are attached to the matrix entries.

[3] This extreme sometimes can be observed if teams claim to be agile without having digested intent and nature of agile practices.

realization levels are taken from two references [8, 15]. The three AMT matrix entries in the figure belong to the requirements viewpoint (Req-0, Req-1) and the functional design viewpoint (Fun-1). User stories, use cases, and component models (specified as a combination of UML class and sequence diagrams) are the selected artifact types in this example.

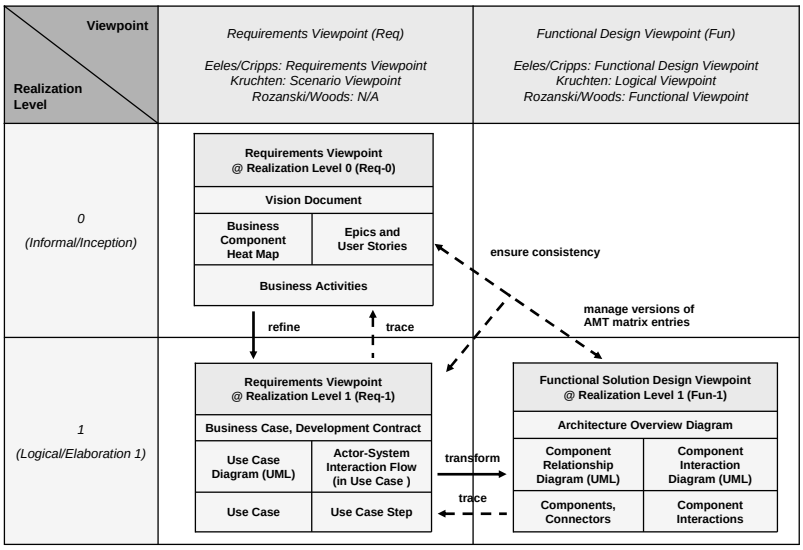

Fig. 2. AMT matrix entry population examples

The AMT matrix for a method produced in steps 1 and 2 answers the following questions:

1. Which viewpoints to use and how many realization levels to foresee (step 1)?
2. Which artifact type(s) to use in each matrix entry, and for what purpose (step 2)?
3. Which notation to select for each artifact and how to customize the selected ones for a particular matrix entry, e.g., syntax profiling (step 2)?
4. Which role to assign ownership of AMT matrix entries to and when in the process to create, read, and update the matrix entries (step 2)?
5. Which techniques and best practices heuristics from the literature to recommend for manual artifact creation and model transformation development (step 2)?
6. Which commercial, open source, in house, or homegrown tools to use for artifact creation, e.g., editors and transformation frameworks (step 2)?

As these questions can be answered differently depending on modeling preferences and method tailoring, the AMT matrix content for a method differentiates practitioner communities (e.g., a practice in a professional services firm or a special interest group forming an open source project).

Having completed steps 1 and 2, the AMT matrix is ready for project use (step 3).

Step 3: Populate a project-specific AMT matrix instance with artifacts. In this step, the AMT matrix structures the project document repository (model space) and is populated with actual artifacts (e.g., models and code) throughout the project.

We continue the presentation of step 3 in the second next section of this chapter. Before that, we formalize the concepts presented so far in a metamodel for artifact management.

3 A Metamodel for the AMT Matrix

In this section, we present a metamodel for the AMT matrix. This metamodel may serve as a reference for tool development.

3.1 Overview

As our approach targets different audiences (i.e., method creators and tool builders, but also project practitioners), we distinguish between an *AMT metamodel*, a *type-level AMT model* and an *instantiated AMT model*. The involved models and diagrams can be summarized as:

1. AMT metamodel, shown in Figure 3.
2. Type-level AMT model, created by a method creator by instantiating the AMT metamodel. Such a type-level AMT model is created in step 2 of our approach.
3. Instantiated AMT model, which is created by instantiating and populating a type-level AMT model when creating project artifacts on a project and categorizing them according to the AMT matrix approach. Such an instantiated AMT model is created in step 3 of our approach.

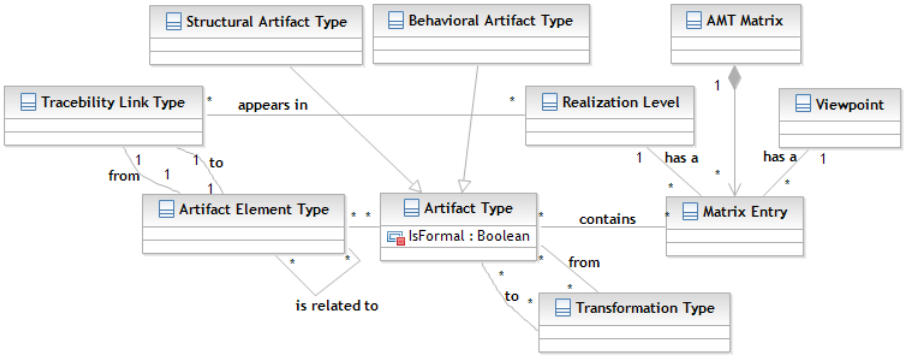

Fig. 3. The metamodel for the AMT matrix (represented as a UML class diagram)

Overall, our organization follows well-established principles that are also used in the Meta Object Facility (MOF) [27]. We now describe the AMT metamodel which is used to formalize an AMT Matrix. We outlined in the previous section that an AMT Matrix consists of several Matrix Entries (expressed by the AMT Matrix class and the Matrix Entry class in the metamodel). Each Matrix Entry represents one combination of a viewpoint and a realization level (e.g., Req-0 or Fun-1) and contains a number of Artifact Types. An Artifact Type can either be a Structural or a Behavioral Artifact Type, e.g., a UML class diagram vs. a sequence diagram. Artifact Types consist of Artifact Element Types such as use cases (or use case steps) and UML classes. Artifact Types together with Artifact Element Types can be considered as the metamodel defining a language (i.e., requirements engineering or architecture design notation), i.e., the UML metamodel for UML class diagrams. UML merely serves as an example here; any other formally defined notation can be represented this way. As a metamodel for a language typically consists of a multitude of related model elements, Artifact Element Types are related to each other by an 'is related to' association.

Each Matrix Entry is given a Viewpoint which categorizes it (e.g., Requirements and Functional Design viewpoints, see Figure 2 in the previous section). Furthermore, each Matrix Entry is associated to a Realization Level which categorizes the artifacts in the Matrix Entry into realization levels such as the two exemplary ones presented in the previous section (i.e., informal and logical); as motivated in the description of step 1 in Section 2, other amounts of realization levels and different names can be defined as well (see Section 5 for examples). An Artifact Type may appear in multiple matrix Entries (e.g., if the same notation is used to create models that serve different purposes).

Traceability between artifacts and their elements is supported by the Traceability Link Type which connects Artifact Element Types. Transformations can be defined as Transformation Types that transform Artifact Types. Examples are use case to component model transformations and links in OOAD/CBD.

The metamodel classes and their associations support the configuration of an AMT Matrix by instantiation, i.e., the creation of a type-level AMT Model. We will now present several examples of creating AMT model instances.

3.2 AMT Matrix Modeling Examples (Applying Step 1 and Step 2).

As a first straightforward example, we configure a matrix that supports the example of the previous section. This matrix consists of two viewpoints, a requirements and a functional design viewpoint. There are two realization levels in each viewpoint. On realization level 0 of the requirements viewpoint, there are four artifact types, Vision Document, Business Component Heat Map, Epics and User Stories and Business Activities. Figure 4 shows the instantiation of the metamodel to express such an AMT matrix configuration. In Figure 4, 'Req' is an instance of the class 'Requirements Viewpoint', which is a subclass of 'Viewpoint' (the subclass is not shown in Figure 3).

A business component heat map indicates areas of a business that require attention and investment because of market dynamics and the current positioning of an enterprise in the market (relative to competition). A component in such a heat map might be home loan processing (in a banking scenario); a related epic and user story set might then be 'modernize and accelerate home loan processing' and 'as a retail bank client, I want to

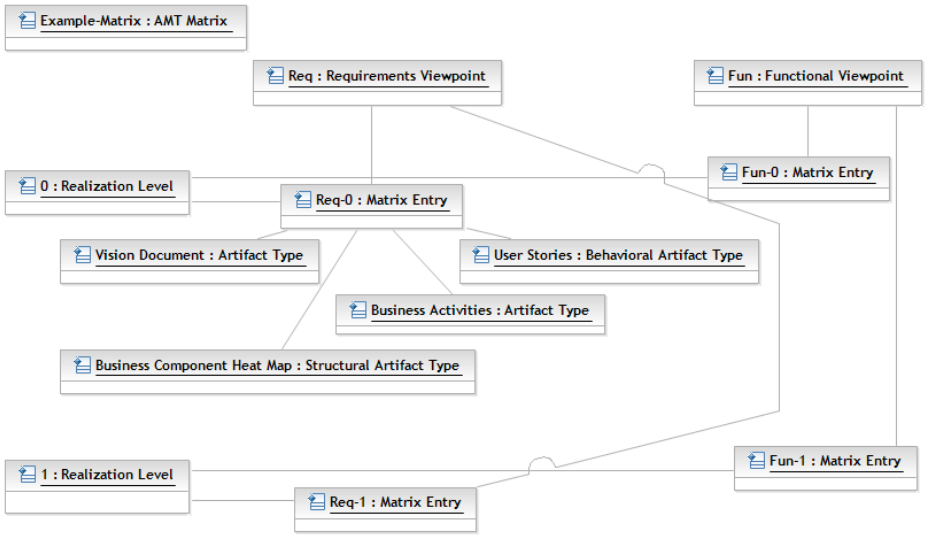

Fig. 4. Example of a type-level AMT model (UML object instance diagram)

be able to apply for a loan online and be able to receive a quote immediately so that I can save time and am able to compare competing quotes'.

These artifacts can be related by transformations that generate parts of artifacts (e.g., an initial set of epics and user stories may be obtained from the heat map). Transformations may also generate traceability links between artifact elements (e.g., to record that a user story has been derived from a heat map).

In addition to the AMT metamodel itself, constraints can be formulated on different levels. Such constraints can pose requirements on a type-level AMT model and also on instantiated AMT models, such as that each configured viewpoint must be represented. Constraints can also be formulated to require a certain relationship in a populated AMT Matrix instance, e.g., that certain traceability links between model elements have to exist. It is also possible to formulate constraints without tying them to a specific AMT Matrix. Such a constraint can state a requirement for an AMT matrix and could require a certain number of artifact types or transformation types to exist.

As a second example, we continue the OOAD/CBD example from step 2 in Section 2 and discuss an AMT matrix configuration where software architects are advised to analyze the use cases to create an initial component model for the solution under construction. We now perform this design activity in an example and derive an AMT Matrix along the way. We decided to apply the component modeling heuristics for this design step from the literature, specifically 'UML Components' by Cheesman/Daniels [7]. This book defines a mature, state-of-the-art OOAD method that is widely adopted in practice by a large population of requirements engineers and architects. Alternatively,

other techniques or heuristics could be applied in this activity as well.[4] With the help of our metamodel, these heuristics can be expressed as model transformations.

In their method, Cheesman/Daniels defined a 'specification workflow' which begins with a step called 'component identification'. Business concepts and use case model are among the input to this step (besides existing interfaces and existing assets). Among other artifacts, it yields 'component specs & architecture', captured in a component model artifact [8]. In our exemplary AMT matrix (Figure 2), use case models reside in entry Req-1 and component models in entry Fun-1. Under 'Identifying Interfaces', Cheesman/Daniels advise to call out user interface, user dialog, system services, and business services components and trace them back to requirements artifacts. These components offer one operation per use case step.

As the next step, they recommend adding a UML interface to the design for each use case step; for each use case, a dialog type component should be added to the architecture. This scheme can be followed by an architect/UML modeler, or partially automated in a tool targeting the analysis-design interface (or, more specifically, the Req-1 to Fun-1 matrix entry boundary in an AMT matrix).

Table 1 maps the AMT metamodel classes from Figure 3 to the artifact types in the OOAD/CBD example from Figure 2 (Section 2). It also lists an exemplary model transformation implementing the Cheesman/Daniels heuristics for the transition from use cases to components:

Table 1. Artifact and Model Transformation (AMT) matrix for OOAD/CBD (developed in step 2 of our approach)

Matrix Entry / **Metamodel concept**	*Requirements Engineering, Level 1 (Req-1)* *Role: Domain Analyst*	*Functional Solution Design, Level 1 (Fun-1)* *Role: Software Architect*
Artifact Type (Structural)	Use case model	Component model (UML class diagram)
Artifact Type (Behavioral)	Use case scenarios	Component interaction diagrams (UML sequence diagrams) for sunny day and error scenarios
Artifact Element Type	Use case Use case step Precondition Postcondition	Components as defined in reference architecture Component responsibilities expressed as initial operations in initial system interface Assertion in component specification Out value of operation plus optional assertion
Traceability Link Type	From component back to use case From operation back to use case step	
Transformation Type	From use cases to functional components (realization level 1): user interface, user dialog, system services, business services	

A full version of this table would provide the complete output of step 2 of our approach for OOAD/CBD.

[4] For instance, domain- and style-specific literature, e.g., on service modeling and SOA design can further assist with this work (see Schloss Dagstuhl Seminar on Software Service Engineering (January 2009) and [29] for examples).

3.3 Additional Metamodel Instantiation Examples and Considerations.

An example of a constraint crossing viewpoints is that no logical functional design component in Fun-1 should exist that does not have any traceability link (existence justification) in a requirement, e.g., a use case (on a lower level of refinement, there might be merely technical utility components that only have indirect links to the business requirements). A second example of a constraint is the rule of thumb that no Req-0 user story (see Section 2) and no Req-1 use case should reference any business entities that have not been defined in a Req-0 glossary or Req-1 OOAD domain analysis model.

Business interfaces (i.e., business type model and business rules) often serve as input to component interface specification work (i.e., providing method signatures). These artifact types can be positioned in our AMT matrix in a similar way as use cases and components; additional traceability link types and transformation types can be defined.

4 Example: Instantiation of Artifact Management for an OOAD Project

Once a type-level AMT model has been created (e.g., the OOAD/CBD one from Sections 2 and 3), one instance of the type-level AMT model is created per project that employs the method described by the AMT matrix (here: OOAD/CBD). This instantiated AMT model is populated by the project team.

AMT matrix instance for Travel Booking System (applying step 3). In the Travel Booking System example in [7], a 'Make a Reservation' use case (for a hotel room as part of a travel itinerary) appears. The use case has the steps 'Identify room requirements', 'System provides price', and 'Request a reservation' (among others). The component identification advice given in the book is to create one dialog type component called 'Make Reservation' (derived from use case) and one system interface component providing initial operations such as 'getHotelDetails()', 'getRoomInfo()', and 'makeReservation()' (derived from the use case steps). Use case model and component model for the Travel Booking System are examples of artifacts; the four use cases, the use case steps, and the two components are examples of artifact elements that are linked with traceability links. The transition from the use case to the initial component model can be implemented as a model transformation. Figure 5 summarizes these examples of model artifacts, artifact elements, traceability links, and transformations.

All component design activities discussed so far take place on a conceptual, platform-independent realization level; in the Cheesman/Daniels method, component realization (covered in a separate Chapter called 'Provisioning and Assembly') represents the transition from realization level 1 to realization level 2 (or from logical to physical design). All design advice in the book pertains to the functional viewpoint; the operational/deployment viewpoint is not covered. Advice how to place deployment units that implement the specified and realized functional components, e.g., in a Java Enterprise Edition (JEE) technology infrastructure from an application server middleware vendor, is available elsewhere in the literature.

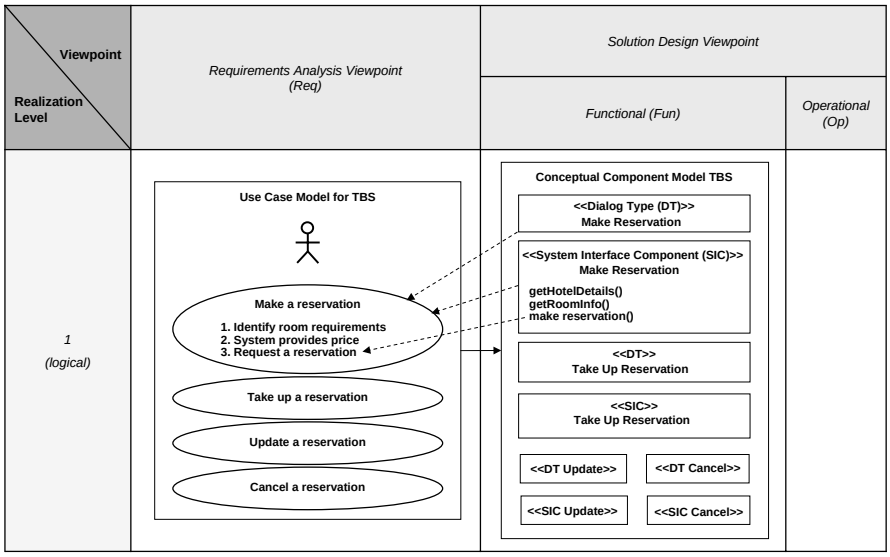

Fig. 5. A part of an exemplary AMT Matrix instance population

Observations in exemplary application of AMT concepts (discussion). The examples in this section demonstrate that our artifact management ideas are in line with those in established methods and recognized text books; our concepts add value to these assets as they organize the artifacts produced and consumed in a structured, interlinked fashion. They also provide a formal underpinning for such assets that tool builders can use to create method-aware tools of higher value than current Create, Read, Update, Delete (CRUD) editors on metamodel instances.

As a side effect, our examples also indicate that it often makes sense (or even is required) to combine methods, techniques on application development and integration projects. An integrated approach to artifact management facilitates such best-of-breed approach to method engineering.

5 AMT Matrix Prototype and Usage Considerations

Prototypical implementation. We have realized AMT matrix support in the *Zurich Artifact Compiler (ZAC)*. ZAC is implemented in Java and Eclipse; its first version has the objective to demonstrate the value and the technical feasibility of our concepts. The current ZAC demonstrator provides Business Process Modeling Notation (BPMN) and UML frontends as well as UML, architectural decisions, and project management tool backends. It is configured with the realization levels and viewpoints from [8] that already served as examples in the previous sections.

The demonstrator supports the following transformations:

- User story to UML use case (Req-0 to Req-1).
- Activity in a process specified in BPMN to UML use case (Req-0 to Req-1, or Req-1 to Req-1[5]).
- UML use case to UML component (Req-1 to Fun-1).
- UML component to architectural decision issues and outcomes (Req-1 to Rat-1).
- User stories to work items in high-level project plan (Req-0 to Mgmt-0), where 'Mgmt' stands for 'Project Management Viewpoint'.

The frontends are implemented as plain Java objects; they process XML files that contain the input models. Backends use Eclipse JET as a template-based transformation framework. The AMT matrix comprises a set of interrelated plain Java objects; the ASTs are realized as graphs, the symbol tables as hash tables.

Figure 6 shows an instance of an AMT matrix for OOAD/CBD (i.e. an instantiated AMT model) as it would be developed on a project. The case study from the previous section serves as example here. This instance was developed with the ZAC prototype; it extends the examples given in previous sections of the chapter. The five viewpoints, three realization levels, and various model artifact types (e.g., component relationship diagram) originate from the *IBM Unified Method Framework (UMF)*. UMF is the standard method used by all IBM architects on professional services engagements with clients; under predecessor names, it has been applied on numerous commercial enterprise application development and integration projects since 1998. UMF leverages the UP metamodel, but adds a rich set of method content.

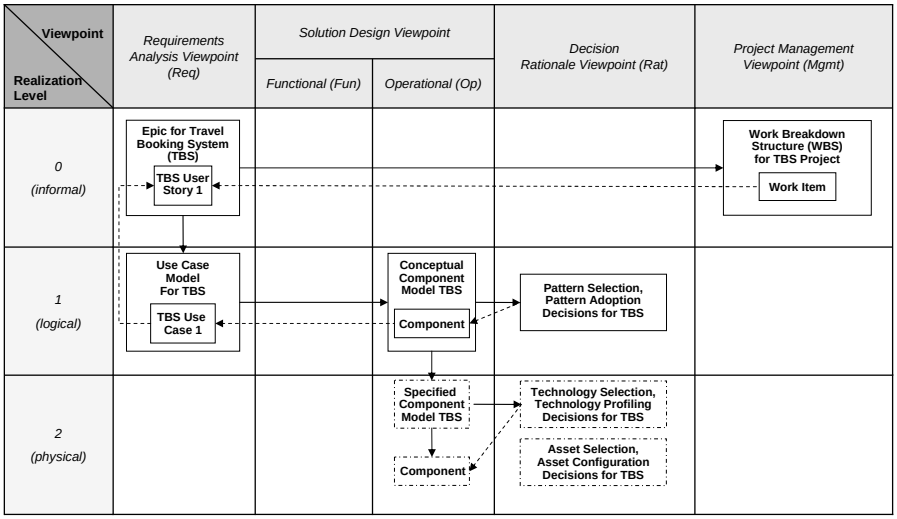

Fig. 6. Exemplary AMT matrix configuration (subset of entries)

[5] depending on the positioning of BPMN in the method used to create and configure the type-level AMT model

In the prototype, an agile user story elicitation effort (Req-0) yields candidate use cases (Req-1), and candidate components (Fun-1); examples of such artifacts appeared in the previous section of this chapter. A work breakdown structure can be derived from these inputs (Mgmt-0); its work items concern the design and development of the candidate components. The transitions between entries (i.e., realization levels or viewpoints) are accompanied by related architectural decisions that preserve the justifications of the chosen designs, e.g., pattern selection and adoption decisions regarding the conceptual component model (Rat-1). The solid arrows in the figure either represent human design activities or model transformations [6]. The dotted arrows represent traceability links. For instance, we have suggested a pattern-centric decision identification technique connecting Fun-1 and Rat-1 in our previous work [32].

Usage scenarios and roles. Software engineering methods can be characterized by their artifacts, their process (i.e., roles, phases/tasks/activities), the techniques for artifact creation they provide, and their reusable content. Hence, we expect method creators to configure AMT matrices according to particular methods such as UP, domain analysts (requirements engineers) to populate the AMT entries for the requirements viewpoint, software architects to own the entries in the functional and operational design viewpoints (i.e., Fun-1, Fun-2, Op-1, Op-2), and platform specialists to populate the design AMT matrix entries on the physical realization level (e.g., developers for Fun-3). These roles may perform the following activities:

1. Populate existing AMT matrices supporting an out-of-the-box configuration (this is a mere application scenario, corresponding to our step 3 from Section 2).
2. Develop custom transformations, e.g., in support of software provisioning; in such transformations, models in the operational viewpoint can be leveraged, e.g., when transitioning from a deployment pattern to a concrete software-as-a-service offering (step 2).
3. Configure AMT matrices with other viewpoint models and/or additional realization levels (step 1).
4. Develop additional applications and utilities leveraging the data in an AMT matrix, e.g., effort estimation applications and artifact search utilities (step 2 and step 3).
5. Instantiate the AMT metamodel to support new methods, artifact management tools, or model transformation frameworks (step 1 and step 2).

Another usage scenario for the matrix is to compare methods and position reusable assets; an agile project uses many realization levels, attempts to produce very few artifacts for each entry, and traverses the matrix several times per day (a formalization of continuous integration); a waterfall project has only one realization level (matrix row) and traverses this row once per project.

Discussion. In the current state of the practice, we find tools that produce a number of artifacts without providing an integrated view of all knowledge about the system under construction. This knowledge is contained in the various created artifacts. Moreover,

[6] such transformations are algorithms/functions that accept one or more models as input and return the same or another set of models as output

some tools overlap in their functionality and expose their users to semantic dissonances and tedious, error-prone import-export or copy-paste activities as it is not clear to which viewpoint and realization level the tool output belongs. Naming conventions and package hierarchies are used to organize artifacts.

These problems can be overcome if configurable AMT matrix support is introduced to existing and emerging requirements engineering and architecture design (modeling) tools to integrate them into an *Integrated Modeling Environment (IME)* that ties in with state-of-the-art development environments such as Eclipse-based Integrated Development Environment (IDEs). Details of such integration effort are out of scope of this chapter (and this book).

Applying our presented approach requires some effort from method engineers and project teams. For instance, matrix dimensions, viewpoints, and realization levels have to be defined and artifact types and artifacts have to be assigned to matrix entries. However, much of this effort is spent anyway, even without our approach being available (if this is not the case, project teams run the risk of creating unnecessary artifacts). Furthermore, well-established engineering practices such as separating platform- independent from platform-specific concerns and distinguishing stakeholder-specific concerns are easily violated if established concepts such as viewpoints and realization levels are not applied. Our AMT matrix approach supports these well-established concepts natively and combines them to their mutual benefit. What is minimally required from the method engineer and the project team is to reflect and clearly articulate the purpose of each artifact type/artifact in terms of its viewpoint and realization levels. We believe that the necessary effort of following our approach is justified by its various benefits: (1) it enables checking the completeness of artifact types/artifacts across all required stakeholder concerns across the software engineering lifecycle, (2) it enables checking the absence of redundancy across artifact types/artifacts and (3) it allows method engineers and project teams to specify and reason about traceability and consistency of artifacts/artifact types in a more conscious and disciplined way.

Furthermore, we believe that the required effort can be minimized through good tool design. For instance, a tool that supports role-specific configuration of its user interface and is aware of method definitions, e.g., of requirements analysis and architecture design methods, can automatically tag artifacts according to their location within the matrix. This assumes that the tool has been configured with an AMT matrix.

6 Related Work

In the software engineering community, viewpoints have been introduced to capture different perspectives of stakeholders involved in the development. The ViewPoints framework [13] introduces a viewpoint to consist of a representation style, a domain, a specification, a work plan and a work record. A viewpoint is used to express the concerns of a particular stakeholder involved in the development. The representation style is used for describing in which language a viewpoint is expressed. The domain is a name given to the part of the world that is seen by the viewpoint. The specification is used to capture a partial system description, using the representation style. The work plan captures the

development process knowledge in this viewpoint and the work record the development history.

The ViewPoints framework has been used in [26] to explore the relationship to a software development process. For that purpose, a viewpoint template is introduced where only the representation style and the work plan slot is fixed. A software engineering method is then a configuration of viewpoint templates and their relationships.

If a software system is described from different viewpoints, this immediately raises the issue of consistency and inconsistency. In the ViewPoints framework, this is addressed by defining relationships between viewpoints, called inter-viewpoint rules. These rules can be checked at certain points in the development process.

In the context of the ViewPoints framework, many case studies have been performed. The ViewPoints framework has triggered substantial research in the area of consistency management (see, e.g., [11, 12, 3, 16, 10]). The work presented in this paper builds on the original work by Finkelstein et al. and extends their concepts in a method engineering context, combining them with realization levels and ideas from model-driven development.

Recent work by Anwar et al. [1] introduce the VUML profile to support view-based modeling from analysis to coding. Each actor of a system is associated with a viewpoint. In each viewpoint, use cases and scenarios are described and class diagrams are derived. A VUML model is then composed of these partial models. This composition is automated using model transformations. In contrast to our work, Anwar et al. focus on model composition for a fixed set of viewpoint models. They do not address the problem of configuring and defining the relationship between different artifacts.

In the software architecture community [24], an architectural viewpoint consists of a 'viewpoint name, the stakeholders addressed by the viewpoint, the architectural concerns framed by the viewpoint and the viewpoint language.' A viewpoint can also include consistency and completeness checks and heuristics, patterns, guidelines. It is explicitly mentioned that a viewpoint is a template for a view. A view itself is an aggregation of models that represents the software system from a specific angle, focusing on some concerns. According to Rozanski and Woods [31], a viewpoint is defined as 'a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views'.

When comparing the different notions of viewpoints one can conclude that viewpoints in the different approaches are very similar. In all cases, a viewpoint consists of stake holders whose concerns are addressed, a notation (representation style or modeling languages), some method or process knowledge, as well as viewpoint relationships. Our work is in line with all presented definitions and combines disjunct/discrete viewpoints with realization levels in a novel way to structure the artifact landscape (i.e., the model space) in a project to the benefit of the various stakeholders. Cross-cutting viewpoints are not yet covered by out AMT formalization.

Another area of related work is tool and model integration. Milanovic et al. [25] describe how artifacts created in a software development process are stored and managed in the BIZYCLE repository. This repository includes repository management operations such as consistency and metadata management and provides central point of integration.

Using a configuration manager, artifacts and their relationships as well as consistency rules can be defined which are then used by the automatic artifact management during a project. The ideas presented by Milanovic et al. are similar, however, their focus is on the common repository. It is not described how such a common repository can be technically realized. Our AMT Matrix concept and the metamodel can be used to structure the common repository when realizing an integrated tool infrastructure. Recent work by Broy et al. [5] criticizes the current state-of-the art of tool integration and proposes different ways to improve the situation. We believe that our AMT matrix can be used as one means and basis for seamless model-based development.

Earlier work in the area of consistency management has already recognized that consistency constraints are dependent on the development process and the application domain. Küster [20] describes a method how consistency management can be defined dependent on the process and application domain. Further, the large body of work on consistency checking and resolution (see, e.g., [17, 9, 23]) must be integrated and adapted by tools working with the AMT Matrix in order to achieve full benefit of it. This also applies to the work on traceability which establishes traceability links based on, e.g., transformations and the work on defining and validating model transformations (see, e.g., [21, 22]).

7 Conclusion

In this chapter, we first observed several problems encountered by method creators and tool builders as well as requirements engineers and architects on projects. We then generalized these problems into conceptual artifact management issues and argued that these issues are currently not sufficiently addressed by techniques and tools from the software engineering and modeling communities. Based on these observations, we introduced the Artifact and Model Transformation (AMT) matrix which provides a categorization and structuring means for artifacts and their relationships in a project. We developed and presented our solution in three steps:

1. Specify AMT matrix dimensions (i.e., viewpoints and realization levels by default).
2. Position method-specific artifact types in AMT matrix entries (according to their purpose).
3. Populate project-specific AMT matrix instance with artifacts (according to their type).

In its default incarnation, our approach combines two commonly applied complexity management concepts, viewpoints and realization levels, to their mutual benefit. Other and/or additional structuring dimensions can be defined (future work). An AMT matrix can be populated with artifact types to support the needs of a specific method; project teams then create and manage artifact instances of these types by their matrix position. Such a configuration requires the definition of relationships and transformations between artifacts as well as consistency and traceability management. We provided a metamodel for the AMT Matrix which can be used as a reference model for integrating AMT matrix concepts into tools. We also presented an exemplary OOAD/CBD instantiation of the AMT metamodel and applied it to a sample project. Finally, we discussed the usage scenarios, benefits, and the implications of our approach.

In future work, we plan to evaluate further our approach, e.g., via personal involvement in industry projects (action research) and via joint work with developers of commercial requirements engineering and architecture design tools. Further research includes the elaboration of how consistency and traceability management can be defined on the basis of the AMT matrix as well as adapting existing modeling tools to support the AMT matrix when defining modeling artifacts. Cross-cutting viewpoints also require further study; tagging the matrix entries that address cross-cutting design concerns such as performance and security to slice AMT instances seem to be a particularly promising direction in this regard.

References

1. A. Anwar, S. Ebersold, B. Coulette, M. Nassar, and A. Kriouile. A rule-driven approach for composing viewpoint-oriented models. *Journal of Object Technology*, 9(2):89–114, 2010.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 2003.
3. E. Boiten, H. Bowman, J. Derrick, and M. Steen. Viewpoint consistency in Z and LOTOS: A case study. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 644–664. Springer-Verlag, Heidelberg, September 1997.
4. G. Booch. *Object-Oriented Design*. Benjamin-Cummings, Redwood City, California, 1991.
5. M. Broy, M. Feilkas, M. Herrmannsdoerfer, S. Merenda, and D. Ratiu. Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments. *Proceedings of the IEEE*, 98(4):526 – 545, April 2010.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture*. Wiley, 1996.
7. J. Cheesman and J. Daniels. *UML Components*. Addison-Wesley, 2001.
8. P. Eeles and P. Cripps. *The Process of Software Architecting*. Addison-Wesley, 2009.
9. A. Egyed, E. Letier, and A. Finkelstein. Generating and evaluating choices for fixing inconsistencies in uml design models. In *ASE*, pages 99–108. IEEE, 2008.
10. G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, 2001.
11. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 84–99. Springer-Verlag, 1993.
12. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.
13. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.
14. M. Fowler. *UML Distilled*. Addison-Wesley, 2000.
15. D. Frankel. *Model-Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

16. C. Ghezzi and B. A. Nuseibeh. Special Issue on Managing Inconsistency in Software Development (1). *IEEE Transactions on Software Engineering*, 24(11), November 1998.
17. I. Groher, A. Reder, and A. Egyed. Incremental consistency checking of dynamic constraints. In *FASE*, volume 6013 of *LNCS*, pages 203–217. Springer, 2010.
18. P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 1999.
19. P. Kruchten, P. Lago, and H. van Vliet. Building up and reasoning about architectural knowledge. In *QoSA*, volume 4214 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2006.
20. J. M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, March 2004.
21. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.
22. J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *Models in Software Engineering, Workshops and Symposia at MoDELS 2006, Genoa, Italy, October 1-6, 2006, Reports and Revised Selected Papers*, volume 4364 of *LNCS*, pages 193–204. Springer, 2007.
23. J. M. Küster and K. Ryndina. Improving inconsistency resolution with side-effect evaluation and costs. In *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2007.
24. M. Maier, D. Emery, and R. Hilliard. Introducing IEEE 1471. *IEEE Computer*, 2001.
25. N. Milanovic, R.-D. Kutsche, T. Baum, M. Cartsburg, H. Elmasgünes, M. Pohl, and J. Widiker. Model&metamodel, metadata and document repository for software and data integration. In *MoDELS*, volume 5301 of *LNCS*, pages 416–430. Springer, 2008.
26. B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Software Eng.*, 20(10):760–773, 1994.
27. Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification, OMG Available Specification Version 2.0, OMG Document Number formal/06-01-01*, January 2006.
28. Object Management Group (OMG). *OMG Unified Modeling Language, Superstructure. Version 2.2., OMG Document Number formal/2009-02-02*, February 2009.
29. M. Papazoglou. *Web Services: Principles and Technologies*. Prentice-Hall, 2007.
30. K. Pohl. *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer-Verlag, 2010.
31. N. Rozanski and E. Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2005.
32. O. Zimmermann. *An Architectural Decision Modeling Framework for Service-Oriented Architecture Design*. PhD thesis, University of Stuttgart, 2009.