Model-supported Software Creation: Towards Holistic Model-driven Software Engineering

Hans-Werner Sehring
Department of Computer Science
Nordakademie
Elmshorn, Germany
e-mail: hans-werner.sehring@nordakademie.de

***Abstract*—Software typically is developed based on descriptions of a relevant section of the real world, the problem as well as its solution. Methodologies and tools have evolved to create and manage such descriptions, and to finally implement software as specified. Model-Driven Software Engineering (MDSE) is one approach of model management. A series of models that build upon each other by means of model transformation is used to describe a software solution in increasing detail. While MDSE gained a fair amount of attention, it is not equally successful in all application domains. We claim that one reason for this is that MDSE is well-suited for formal domains and computation-centric solutions. But it is not equally suited for software development processes with a high degree of creativity involved, like, for example, solutions with a focus on human-machine interaction or content-centric applications. One reason is the fact that properties of such software are designed by experts of certain domains who use specific notations and tools. In this paper, we discuss an approach for the creation of software that requires models that are either defined in specific notations used by experts or that do not allow formalized model transformations. The approach relies on artifacts created using a heterogeneous set of languages. These artifacts are described by formal models that add semantics and that relate the informal artifacts. For such an approach, we coin the term "model-supported software creation" in this paper.**

***Keywords*—model-driven software engineering; model-driven architecture; software engineering; software architecture**

I. Introduction

Software is, in most of the cases, used to represent and solve real-world problems. In order to be able to do so, a relevant section of the real world needs to be captured, and the problem as well as its solution need to be described in sufficient detail. This includes defined requirements, test cases, conceptual models, domain models, etc.

Methodologies and tools have evolved that capture problems and solutions, model the real world with respect to the problem at hand, and finally allow implementing software as specified.

The various description artifacts involved in software engineering processes call for means to manage these descriptions. In particular, they have to be related to each other to reach goals like, for example, those of coherence and traceability.

Classical software engineering has a typical sequence of an analysis phase, resulting in requirements, design phases, resulting in solution designs, and implementation phases, resulting in working software. In agile approaches, these phases may be very condensed. The artifacts (descriptions, models, code, etc.) created in each phase build upon each other. Still, they are formally unrelated. Those artifacts contributing to a phase consider the artifacts from previous phases, though.

Model-Driven Software Engineering (*MDSE*) or *Model-Driven Software Development* (*MDSD*) is one approach to a more formal management of artifacts. A series of models that build upon each other is used to describe a software solution in increasing detail. Typically, the models are refined or transformed up to the point where actual running software can be generated out of the most precise model.

While MDSE gained a fair amount of attention, it is not equally successful in all application domains. We claim that one reason is that MDSE is well-suited for formal domains and computation-centric solutions. But is is not equally suited for software development processes with a high degree of creativity involved. For example, while it is feasible to model technical domains, for example, involving mathematics and physics, it is less common to formally model solutions with a focus on creative and subjective aspects. Human-machine interaction (online shops, for example) or content-centric applications (personalized marketing websites, for example) are examples found in typical customer-facing commerce systems.

For models that experts require in specific notations, and for ones that do not allow formalized model transformations, a different approach is discussed in this paper. It relies on models created using a heterogeneous set of languages that are described by formal models that add semantics and that relate the informal artifacts.

We introduce the name *Model-Supported Software Creation* (*MSSC*) in this paper to emphasize the fact that (formal) models are supporting a creative process, but are not the central resource of the process, and to describe the wider range of activities involved.

Section II of this paper revisits some approaches to MDSE. Shortcomings of simple MDSE applications are examined in Section III. Requirements to a holistic MSSC approach are listed in Section IV. Section V presents the *Minimalistic Meta Modeling Language* (*M³L*) and how it is applied to holistic MSSC. We conclude the paper in Section VI.

II. Model-driven Software Engineering

Various approaches to software generation from models are discussed. In this section, we briefly revisit some of these.

A. Model-driven Architecture

The *Model-Driven Architecture* (*MDA*) [1] of the Object Management Group (OMG) is an early and well received proposal for an MDSE approach. It assumes models to be created on (originally) three levels of abstraction. A *Computation-Independent Model* (*CIM*; this term is not used in current specifications) describes the software to be developed from the perspective of the subject domain, as domain concepts or requirements. It typically is an informal description, for example, done in natural language. A first formal model is a *Platform-Independent Model* (*PIM*), formulated in the MDA's *Meta Object Facility* (*MOF*). It is transformed into a *Platform-Specific Model* (*PSM*) that in turn is used to generate a working implementation.

B. Software Generation

Software generation has gained particular attention since this step in an MDSE process can well be formalized.

a) Metaprogramming: Programs that generate programs are an obvious means to software generation. The development of such generators tends to be costly, but results may be targeted optimally to the application at hand.

b) Templates: Code with repeating structures can be formulated as templates with parameters for the variations of that uniform code. For *Concept-Oriented Content Management* [2], for example, code for CRUD operations is generated. This code does not differ in functionality, but in the data types used for domain entities.

c) Generative AI: The currently emerging generative AI approaches based on large languages models provide another means to generate code from descriptions. Based on a library of samples, they allow interactively generating code from less formal descriptions, in particular natural language expressions.

C. Domain-specific Languages

Languages can be associated with metamodels [3]. This means that a model of a software application can be expressed by a language for a subject domain. Such a language is called a *Domain-Specific Language* (*DSL*).

The software generation process is simplified to defining an application using a DSL, allowing to define the application in terms of the subject domain. There is a trade-off regarding the degree of abstraction: The more domain knowledge is put into the DSL, the simpler it is to define an application. But a more specialized DSL also means that the range of application that can be defined becomes more limited.

D. Generic Software

The aim of MDSD and MSSC is custom software that is tailored to solve one specific problem. Generic software, on the other hand, encapsulates some domain knowledge that is applicable in a set of scenarios.

The concrete application is defined by setting parameters of the generic software. The application areas of generic software are defined by the degree to which domain knowledge was generalized and parameterized.

There are varying degrees of parameterization. This relates to so-called low code and no code approaches. These are also based on a generalized software that maps a section of the real world, and they allow software to be customized within the limits of the chosen section.

III. MDSE in Practice

MDSE approaches are not equally successful in all application domains [4]. We see two main obstacles to applying MDSE in some areas: heterogeneous modeling artifacts and the stages of a software development that are covered.

A. Heterogeneous Modeling Artifacts

MDSE typically is based on a modeling framework that supports all stages of a software development process. This requires that model artifacts on every stage can be expressed in a language that is supported by that framework. In many cases, it is even required that all models involved are formulated within the same metamodel.

Some application domains call for specific kinds of artifacts that rely on certain established notations and cannot be forced into a form given by a central metamodel. For such application domains, the properties of software are designed by experts of certain fields who use specific notations and tools. One example of such an application domain is digital communication like marketing and sales communication over a website.

In the retail sector, for example, we note that customers interact with retail companies at different touch points, interact on changing communication channels, use different payment methods, are subject to different legal and tax systems, etc. In such scenarios, a series of experts needs to gather (a part of) the domain knowledge on one modeling stage in order to communicate it to experts of the next stage (domain expert to requirements engineers, these in turn to architects as well as test engineers, architects to developers, and so on).

User experience designers and user interfaces designers, for example, work with artifacts like personas, customer journeys, wireframes, style guides, click dummies, prototypes, etc. Such artifacts support creative processes. They are adequate means to communicate with business experts, and they are used by programmers to build usable software.

A pure MDSE approach of generating such artifacts from models is not adequate for the work of experts and their clients. It might be hindering the creative process.

B. Coverage of all Project Stages

Modeling starts at the point where there is consensus about the kind of software to be developed. In fact, projects start at an earlier stage at which a (business) need arises. In a commercial setting, this may be, for example, increased revenue, a certain number of new customers, or customer satisfaction. A solution approach is not given. At this stage, it is not even decided that new or improved software will be part of the solution.

The same holds for project stages after software generation, namely roll-out, operations, and support.

TABLE I. STAGES OF SOFTWARE CREATION

Creation stage	Model entities on the stage
(Business) Goals	KPIs OKRs
Subject domain model	Information architecture Interaction design Wireframes Processes, data flows
Requirements	Solution hypothesis Functional ~ Non-function ~ Customer journeys Touch points
Solution architecture	Interfaces High-level architecture Functional mapping
Software architecture(s)	Components communication between those components interfaces to the environment constraints of the resulting software system requirements met by the architecture rationale behind architecture decisions
Code	Metaprogramming Software generators Domain-specific languages
Systems architecture	Infrastructure definition Automated deployments
Operations	Service level agreement Monitoring

IV. HOLISTIC MODEL-DRIVEN SOFTWARE CREATION

In Section III, we pointed out two shortcomings with basic MDSE approaches: Firstly, they do not consider early project stages that precede software development. Secondly, they are not suited to utilize heterogeneous models that are formulated in different languages, are not all equally formal, etc.

As noted in the introduction section, we use the term MSSC to describe a holistic approach to software creation that in contrast captures all aspects of a project, not only the software development phases, and that can cope with heterogeneous and informal modeling artifacts.

In the following, we point out the modeling stages we consider relevant for software creation processes, and we outline typical model transformations of model-based development processes.

A. Modeling Stages

Table I gives an overview over typical stages of software creation and some examples of artifacts they deal with.

a) Business Goals: A project starts with the identification of a problem to be solved. In many cases, the problem does not lie within the computing domain. Accordingly, the desired solution is typically formulated by means of (business) goals that shall be reached (see Section III-B).

Goals have to be measurable in order to judge the success of a project. *Key Performance Indicators* (*KPIs*) or *Objectives and Key Results* (*OKRs*) are often used to define target values that can be measured. The values that are measured often lie in the business domain and have to be determined by controlling means on the business level. The success of a software solution that helps reaching the goal is then proven implicitly.

Since formal goals are set up as a first abstraction of the business goals to be reached, they are subjective and depend on a stakeholder who defines them. Approaches like i* [5] aim to model this subjectivity.

b) Subject Domain Model: The later stages of software design require a certain understanding of the problem domain, for example, typical concepts of the area the software is to be applied in. The requirements relate to the domain concepts.

Modeling means abstracting from the domain that is represented. Therefore, domain concepts cover a section of the subject domain that is relevant for the solution.

In the MDA approach, the CIM may include the stage of domain modeling.

c) Requirements: Requirements characterize the properties of a software solution. This means that this stage only is entered if it is decided that software helps reaching the defined goals. It also means that a first software solution hypothesis has been recognized and is being detailed through requirements.

There is a wide range of requirements: functional requirements and the diverse kinds of non-functional requirements. Additionally, constraints that limit the solution space belong to this stage.

Other entities of this modeling stage depend on the problem at hand. For example, conceptions of interactive applications for digital communication typically begin by identifying personas as role models of target groups, determine the customer journeys as the sequence of interactions users have at different touch points, before finally deriving artifacts like the information architecture. To design user interfaces, artifacts like wireframes, style guides, and click dummies are used to help defining subject domain concepts and requirements.

There are various tools to help managing functional requirements. Deductive databases can help validating and completing requirements [6].

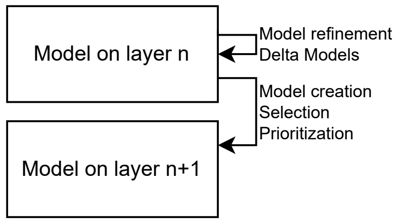
d) Solution Architecture: Solution architecture is the set of high-level definitions that relate subject domain concepts to technical solutions.

As a high-level architecture, it does not prescribe an actual implementation in full detail. It may contain the choice for certain implementation technologies and products, though, in particular if they are crucial to meeting some requirements or to conform to the constraints.

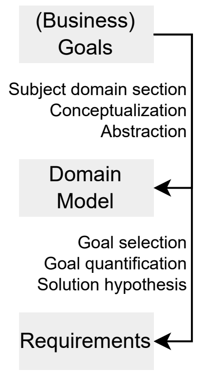
Based on the chosen components, a solution architecture defines the interfaces required to implement the processes and data flows identified as requirements. For example, in a digital communication like an e-commerce website, the information demand at every touch point is derived from the customer journeys, and data flows are designed accordingly.

e) Software Architecture: The detailed design of the software to be developed is part of the software architecture. It details definitions from the solution architecture up to the point where they are concrete enough to guide the coding stage.

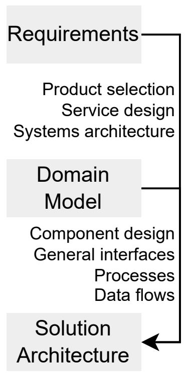
Shaw and Garlan [7] point out that there are different approaches to the different perspectives on software. In a structural approach, the software architecture is composed of components, communication between the components, product



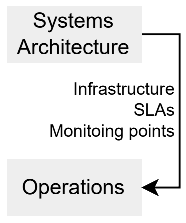
(a) General model transformations



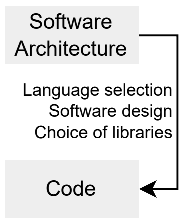
(b) Model transformations for subject domain model



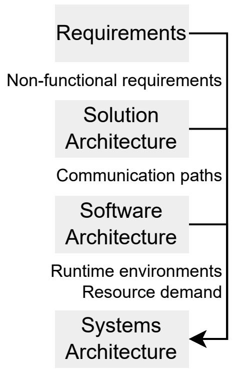
(c) Model transformations for solution architecture



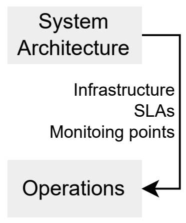
(d) Model transformations for software architecture



(e) Model transformations for code generation



(f) Model transformations for system architecture



(g) Model transformations for operations

Figure 1. Different kinds of model transformations.

configurations, references to the requirements and constraints from the requirements stage, boundaries within which the software is designed to work as specified, the rationale of design decisions, and design alternatives that were considered.

Many other architecture definitions contain similar modeling entities. *Architectural Description Languages* (*ADLs*) allow capturing these aspects.

Shaw and Garlan point out that besides structural models, there are also framework models, dynamic models, and process models. The latter, for example, focus on the dynamic aspects of the software.

f) Code: When architecture models are precise enough, code can be generated out of them using one of the approaches from Section II-B.

In practice, coding is a manual task in most cases. The architecture definition serves as a guideline to programming, documentation, and quality assurance. Detailed design decisions are added in the coding stage.

g) Systems Architecture: The systems architecture describes how software is deployed and set up. It defines computing and communication infrastructure.

Deployment diagrams describe how software is packaged and distributed on the infrastructure. Infrastructure and network diagrams illustrate the technical setup.

Typically, infrastructure is virtualized and created automatically from scripts in the *Infrastructure as Code* approaches. This allows continuous deployments of many software components, for example, in contemporary composable architectures.

h) Operations: Part of the requirements are typically formulated towards operations. *Service-Level Agreements* (*SLAs*) define measurable goals to systems operation. Fulfillment of these goals is controlled by means of monitoring and timely maintenance in the case of incidents. To this end, monitoring and logging concepts connect development and operations.

B. Model Refinement and Transformations

An MDSE process relies on a series of models where models are created from existing models by means of *model transformation*. A model on one stage is created based on the input of models of earlier stages or by refining models from the same stage. There are three typical kinds of model transformations.

Figure 1a shows the basic structure of model transformations on one stage and between stages. Figures 1b to 1g show examples of typical model transformations between different stages.

a) Model Combination: Domains often rely on base domains. For example, business tasks rely on mathematics. Accordingly, models are defined by integrating (existing) models of the base domains. This way, models are reused.

b) Model Refinement: Within one stage, models are refined to more concrete models of the same stage. This way, the work in each stage starts with first, coarse-grained models, that are then transformed into more concrete models. Different refinements of one model may cover different perspectives on the (software) solution. The process of refining involves decision making. Decisions can be documented by explicitly stating delta models that explicitly represent the refinements.

c) Model Creation from Existing Models: When processing from one stage to another, initial models are required for the subsequent stage that is entered. These models shall be related to the most concrete models of the preceding stage. In some cases, models can be transformed when proceeding to a subsequent stage. In this case, the transformation establishes the relationship. If new models have to be created, the model elements should be explicitly linked to the elements from models on which they are based. For example, Shaw [7] demands that a software architecture description refers to requirements.

V. AN MSSC APPROACH WITH THE M³L

An MSSC approach includes the creation and utilization of diverse artifacts. Each of them serves a specific purpose, and each is maintained by experts using established tools. Though the artifacts from different stages of a software creation process are related, they typically cannot be expressed using the same language. They differ, for example, in the level of detail, the degree to which they follow a formalism, and the syntactic representation targeted at different audiences.

When, in contrast to MDSE, no single modeling language can be used for a universal model, an overarching modeling framework is required for model coherence [8]. Such a framework cannot host the artifacts themselves. It shall, however, put the artifacts in context and relate them to each other.

Relationships between artifacts clarify their contribution to the software creation process. They explicate the provenance of models, they put models in context, and they are the basis for traceability and, therefore, the ability to cope with change.

In this paper, we propose using the *Minimalistic Modeling Language*, *M³L* (pronounced "mel") [9], as the modeling framework required for MSSC.

A. A Brief Introduction to the M³L

The M³L is a meta modeling language. As such, it can be employed for models for different kinds of applications.

In this section, we give a brief overview over the syntax of the language. Sample applications in the subsequent sections will demonstrate its use.

A statement **A** defines or references a *concept* named *A*. The M³L does not distinguish definitions from references. If **A** does not exist, it is defined.

Concepts can be *refined* with "is a": `A is a C`. Using the clause "is the" defines a concept to be the only specialization of its base concept.

Concepts can be put in *context*. A statement `A { B }` defines *B* in the context of *A*. *B* is said to be the *content* of *A*. References are valid in the context they are defined in and in all subcontexts. This means, that statements `A { B }` and **C** make *B* and *C* visible in the context of *A*, but *B* is not part of the content of *C* or of the topmost context.

Concepts can be defined differently in different contexts. For example, the statements `A { B is a C }` and **B** define *B* as a specialization of *C* in the context of *A*, but without base concept in the topmost context.

A concept in a nested context is referenced as `B from A`.

Semantic rules can be defined on concepts, denoted by "|=". A semantic rule references another concept that is delivered when a concept with a semantic rule is referenced. Like for any other reference, a non-existing concept is created on demand.

Context, specializations, and semantic rules are employed for *concept evaluation*. A concept evaluates to the result of its syntactic rule, if defined, or to its *narrowing*. A concept *B* is a narrowing of a concept *A* iff

- *A* evaluates to *B* through specializations or semantic rules, and
- the whole content of *A* narrows down to content of *B*.

To evaluate a concept, syntactic rules and narrowing are applied repeatedly.

With this evaluation, for example, a conditional statement can be defined as (given *Statement*, *Boolean*, *True*, and *False*):

```
IfThenElse is a Statement {
  Condition is a Boolean
  ThenStmt is a Statement
  ElseStmt is a Statement }
IfTrue is an IfThenElse {
  True is the Condition } |= ThenStmt
IfFalse is an IfThenElse {
  False is the Condition } |= ElseStmt
```

Concepts can be marshalled/unmarshalled as text by *syntactic* rules, denoted by "|-". A syntactic rule names a sequence of concepts whose representations are concatenated. A concept without a syntactic rule is represented by its name. Syntactic rules are used to represent a concept as a string as well as to create a concept from a string.

For example, rules for language-dependent code generation:

```
Java{IfThenElse |- "if" "(" Condition ")"
                   ThenStmt FalseStmt  .}
```

B. Dimensions of Model Relationships

The three model relationships named in Section IV-B can be expressed with the M³L. This way, models are put in context. The following examples outline basic modeling approaches for the three relationships.

a) Combining models: For example, on the layer of domain models, a model

```
ProductDescriptions is a DomainModel {
  ProductData
  PaymentMethods from Commerce
  PackagingInformation from Logistics }
```

combines parts of product details that come from different specialized models (assuming that concepts for models *Commerce* and *Logistics* are given).

Likewise, on the layer of solution architecture, a model

```
OurInfoSys is a PlatformIndependentModel {
  AppServer from SWComponents
  DBMS from SWComponents
  DataSchema from DBModeling
  WebServer from SWComponents
  WebPage from WebDesign }
```

combines technical components from different technical descriptions.

b) Refining models: One model can be created as a refinement of another. Concepts in the content of the refined model are inherited and can be refined further.

An example from the solution architecture layer is:

```
OurInfoSysConcept is an OurInfoSys {
  RDBMS from SWComponents is the DBMS
  ProductDataSchema
    is an RDBSchema from DBModeling,
       the DataSchema
  WebServer from SWComponents
    is a ServletEngine from Java }
```

In this example, two aspects of the conceptual model are refined: From a technical perspective, the *DBMS* is more concretely specified to be a relational DBMS (*RDBMS*), and the *WebServer* to be implemented as a Java Servlet engine (*ServletEngine*). Regarding the domain model, it is defined that the data schema is defined to store products (*ProductDataSchema*).

c) Creating models in subsequent stage: A model can be explicitly created as a transformation of another model using a semantic rule. In the example of the information system:

```
OurInfoSysConcept |= OurInfoSysDataLayer {
 RDBMS
 ProductDataSchema {
  ProductsTable is a Table from DBModeling
 } }
```

RDMBS from the source model *OurInfoSysConcept* is re-introduced in the transformed model. The database schema *ProductDataSchema* is additionally redefined by naming one table. *WebServer* from *OurInfoSysConcept* is not considered in the transformed model, since it only models the data layer of the information system.

C. Software Creation with the M³L

The models in MDSE ultimately reach the stage of generating code. The M³L allows creating code using syntactical rules that can be added to models with sufficient concreteness.

Using the example from above, part of the information system based on a relational database can be defined to create a relational schema by SQL statements as follows:

```
OurInfoSysDBIm is an OurInfoSysDataLayer {
 ProductDataSchema {
  ProductsTable |- "PRODUCTS("Columns")" .
 } |- "CREATE TABLE " ProductsTable . }
```

By defining the syntactical rules in the context of an implementation model, different code generation schemes can be defined for one software model.

VI. Summary and Outlook

This section sums up this paper and outlines future work.

A. Summary

In this paper, we revisit MDSE approaches and conclude that they are successful in certain application areas, while they are not established in many other areas. In particular, in digital communication, for example, in the construction of commerce or marketing websites or mobiles apps, they are not used in practice. One reason for this is a mismatch between established means of conceptual work and formal models.

Under the name of Model-Supported Software Creation (MSSC) we study requirements to models for such kind of applications. As early results, MDSE approaches cover the stages of software creation well, but they do not cover early inception phases. We claim that models used in MSSC need to be able to cope with less formalism and preciseness as required by typical MDSE approaches. Instead, they must deal with heterogeneity and subjectivity.

We outline model creation with the M³L as a step towards MSSC. It allows providing descriptive models of the artifacts used in practical approaches and relating them as to drive holistic software creation processes.

B. Outlook

We are at the beginning of our investigations towards MSSC. Consequently, there are numerous questions to be answered in the future. We highlight two of them.

There are numerous approaches to generate code from models, and code written in a formal language can be managed in a structured way. The syntactic rules of the M³L, for example, allow this. To include artifacts from other stages into the modeling process (like requirements or design documents), abstractions are needed to reference, include, or generate parts of artifacts the same way it is possible for code.

Testing is typically not found in model-based processes. Though there may be no need to test generated software, a kind of testing is required, nevertheless. This may include model checking on each stage of the process and analysis of models that are the result of model transformations.

In MSSC processes, success should ultimately be judged based on the degree to which business goals have been reached. To this end, they must be formalized, and effects of the running software need to be measured.

Acknowledgment

Numerous discussions on topics of modeling and software engineering led with colleagues, partners, and clients are highly appreciated. The author thanks the Nordakademie University of Applied Sciences for funding the publication of this work.

References

[1] Object Management Group. *Model Driven Architecture (MDA)*, MDA Guide rev. 2.0, OMG Document ormsc/2014-06-01, [Online] Available from: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01. 2023.9.5.

[2] H.-W. Sehring, S. Bossung, and J. W. Schmidt, "Content is Capricious: A Case for Dynamic System Generation," Proc. 10th East European Conference (ADBIS 2006), Springer, 2006, pp. 430-445.

[3] T. Kühne, "Matters of (Meta-) Modeling," Software & Systems Modeling, vol. 5, pp. 369-385, Dec. 2006.

[4] J. Cabot, R. Clarisó, M. Brambilla, and S. Gérard, S., "Cognifying Model-Driven Software Engineering," Proc. Software Technologies: Applications and Foundations (STAF 2017), Springer, 2018, pp. 154-160.

[5] E. S. K. Yu and J. Mylopoulos, "From E-R to "A-R" – Modelling strategic actor relationships for business process reengineering," Proc. 13th Int. Conf. on the Entity-Relationship Approach (ER'94), Springer, 1994, pp. 548-565.

[6] H. W. Nissen, M. A. Jeusfeld, M. Jarke, G. V. Zemanek, and H. Huber, "Managing multiple requirements perspectives with metamodels," in IEEE Software, vol. 13, no. 2, pp. 37-48, March 1996.

[7] M. Shaw and D. Garlan, "Formulations and Formalisms in Software Architecture," Computer Science Today: Recent Trends and Developments, Lecture Notes in Computer Science, vol. 1000, pp. 307-323, 1995.

[8] S. Bossung, H.-W. Sehring, M. Skusa, and J. W. Schmidt, "Conceptual Content Management for Software Engineering Processes," Proc. Advances in Databases and Information Systems, 9th East European Conference (ADBIS 2005), Springer, 2005, pp. 309-323.

[9] H.-W. Sehring, "On Integrated Models for Coherent Content Management and Document Dissemination," Proc. 13th International Conference on Creative Content Technologies (CONTENT 2021), 2021, pp. 6-11.