# Model-Supported Software Creation: Extending Model-Driven Software Engineering with Non-Formal Artifacts and Transformations

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

***Abstract*—Software typically is developed based on descriptions of a relevant section of the real world, the problem at hand as well as the software to be built for 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 the application domain and the software solution under consideration are reflected by such models, other aspects of a software project are not always considered on equal level. Examples are the business side of the project that usually exceeds the software creation part, creative activities like user interface design, and aspects of the operation of the software. In this article, we discuss aspects of extending MDSE towards a holistic approach that includes additional phases of software engineering and the incorporation of models that are either defined in specific notations used by experts or that do not allow formalized model transformations. The approach relies on artifacts that are 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 article.**

***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 with respect to such a model.

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.

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.

*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.

Software engineering, and thus MDSE, at best captures the whole software lifecycle. Ultimately, all development steps are captured by a *holistic* MDSE approach. In this article, we study two dimensions in which to extend typical MDSE approaches. We use the name *Model-Supported Software Creation* (*MSSC*) for an accordingly extended kind of MDSE. This article is an extension of the presentation of the first ideas towards holistic software (project) models in [1].

While MDSE gained a fair amount of attention, it is not equally successful in all application domains [2]. We see two main obstacles to applying MDSE in some areas: the heterogeneity of modeling artifacts and the stages of software development that are covered by a software engineering process.

1) MDSE is well-suited for formal domains and for computation-centric solutions. But is is not equally well applicable to software development processes with a high degree of creativity involved. For example, it is feasible to model technical domains that are based on mathematics and physics. But it is less practical 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.
2) MDSE focuses on the stages of a project where the actual software is specified and implemented. Most approaches start with defined requirements. Projects include more tasks than just software creation alone. There are earlier stages in which (business) goals are set and a decision is made to start a project, and there are stages that follow software implementation, like operations and maintenance.

In order to incorporate these aspects in MDSE, we study a modeling approach that allows incorporating models in

varying notations and modeling approaches that do fully rely on formalized model transformations. It is based on models that are created using a heterogeneous set of languages and that are used to add semantics to and that relate informal artifacts.

Section II of this paper revisits some approaches to MDSE. Additional demand for modeling that exceeds software modeling is studied in Section III (extended demand for additional project phases) and in Section IV (requirements to the integration of both formal and informal models). Section V presents the *Minimalistic Meta Modeling Language* (*M³L*) that we use as the basis for first experiments with holistic MSSC models presented in Section VI. We conclude the paper in Section VII.

## 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*) [3] 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. Model transformations are specified using *Query View Transform* (*QVT*) based on MOF.

### 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* [4], 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 [5]. 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 MDSE 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 for the Full Software Lifecycle

On top of software models as provided by approaches like the ones presented in the preceding section, there are additional aspects of (software) projects that have to be captured in a holistic software engineering process. In this section, we outline typical project activities and intermediate results.

Further artifacts play a role for these aspects, and they call for additional model contributions.

We call sets of project contributions that logically belong together a *modeling stage*. This term shall reflect the fact that models build up upon each other, and not in a temporal sense as terms like *project phase* would indicate.

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

### A. Business Goals, Project Goals, and Constraints

The purpose of software typically is not just to be useful by meeting the requirements, but it contributes to some business goals. At least in commercial applications, business goals are defined upfront, and software may be one part of a solution to reach these goals.

Software modeling in the MDSE sense 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 some degree of 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.

Therefore, on top of a software project, a business endeavor is pursued. The business goals will finally be the main criteria to measure project success. To this end, these goals need to be precisely formulated so that their impact on the software

TABLE I. STAGES OF SOFTWARE CREATION

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

models becomes apparent. Furthermore, the degree to which goals are met needs to be measurable. This calls for a business model to be formulated at the very beginning of a model-supported software creation project.

A project starts with the identification of a problem to be solved as a contribution to a business goal. In many cases, the problem does not lie within the computing domain. Accordingly, the desired solution is typically formulated by means of (project) goals that shall be reached.

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 used to define target values. 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, assuming that it substantially contributes to reaching the business goals.

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* [6] aim to model this subjectivity.

### B. 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. Together, they form a first model of a software solution that, however, is typically informal.

Additionally, (project) constraints that limit the solution space belong to this stage.

There are various tools to help managing functional requirements. Deductive databases can help validating and completing requirements [7].

### C. 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.

### D. Creative Tasks in Software Development

Requirements can be defined in different ways. In requirements engineering, one aims at specifying properties of the software to be built with adequate precision. In agile approaches, requirements are formulated from a business perspective.

The formal models underlying the model-driven approaches as discussed in Section II require abstraction capabilities for viewers to imagine the software to be built and how it will meet the requirements. The ability to work with such abstractions cannot be taken for granted for domain experts and various stakeholders in a software project.

Modern development approaches, in particular agile ones, are based on the engagement of stakeholders, though, requiring them to understand the outcome of each development step. To this end, often visual communication is used for participants who are not comfortable with working with abstract representations of software as used by software developers in an MDSE process. This visual communication is based on the creative input contributed by a cross-functional team. Such creative input is usually found in 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.

Creatively working team members and certain other domain experts prefer using specific notations and tools. Creative tasks are typically carried out on the basis of visual presentations that lack formal completeness and soundness. Still, they help develop a common understanding between the various participants in a project and are, therefore, central to the communication with stakeholders and domain experts to allow them to judge the ongoing development on a subjective basis.

Formal models are well-suited to reason about the emerging software solution, but not about the inputs that led to its design.

For all these reasons, informal means of communication between creative workers, analysts, etc. and stakeholders in a *user-centered development process* typically cannot be substituted by abstract models of software alone.

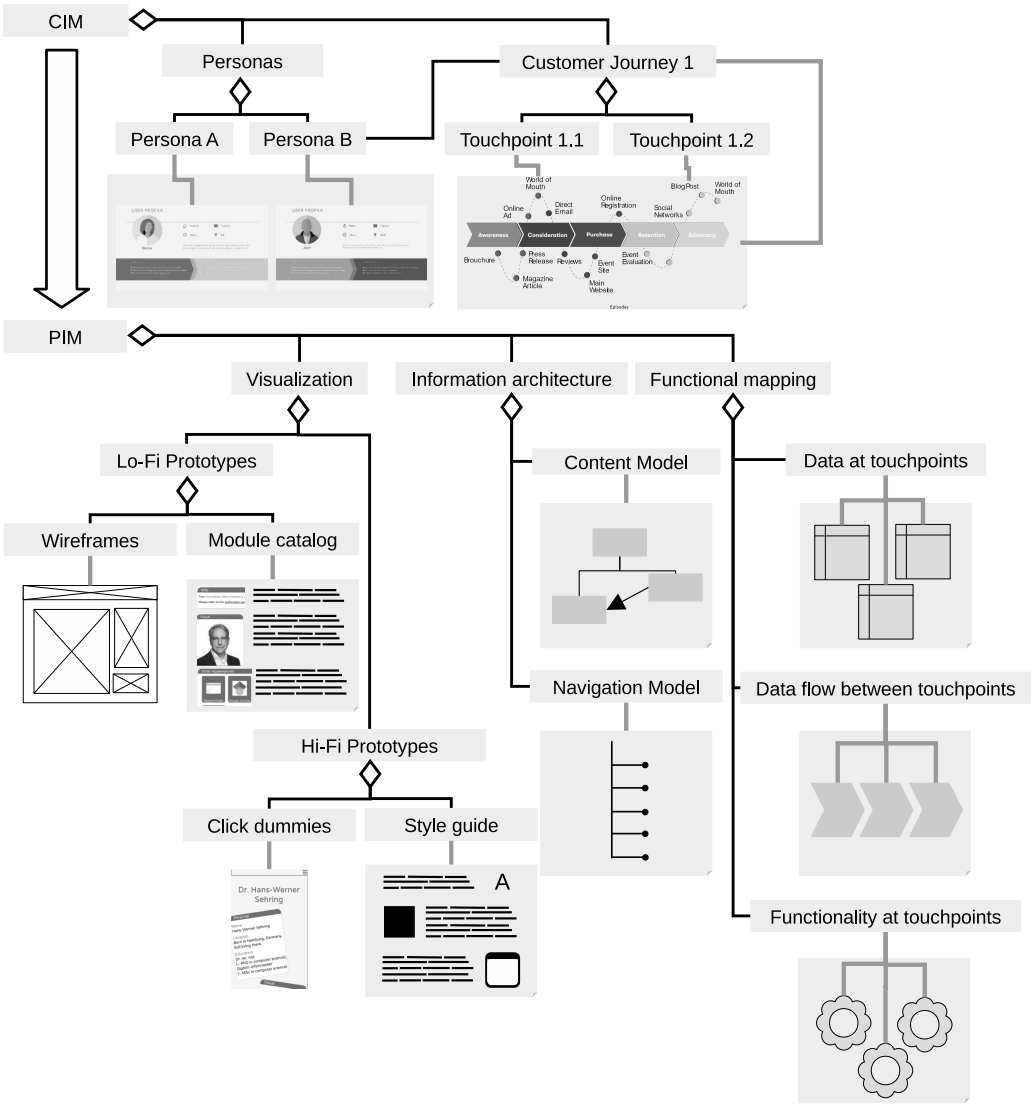



Figure 1. Creative tasks in software development processes.

In MDA terms, a CIM is elaborated to a "visual PIM" that describes the software from the perspective of users (in contrast to the developers' perspective). In this PIM, there is an emphasis on the user experience and the visual appearance of software. This model defines how users of a service interact with the provider of that service, for example, how customers interact with an online shop.

Figure 1 sketches a typical user-centered development process, with the sequence of models shown at the left of the figure.

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 and sketches provide first impressions. Later, graphical details are added in documents like style guides. Click dummies allow stakeholders to get an impression on how the software is intended to be operated.

All these conceptual and graphical descriptions are, on the one hand, defining user experience aspects like navigation and visual layout. On the other hand, they contribute to the definition of subject domain concepts and requirements.

This phase of creating a visual PIM is not concerned about software implementation. Feasibility studies for the realization of the design in software should guide the creative activities, though.

Software design is added at a subsequent stage by creating a PIM in the sense of the MDA. Starting from this point, the steps of typical MDSE processes follow.

The transition from a visual, user-centric PIM to a software-centric PIM depends on the descriptions used. If the aforementioned typical results like personas and customer journeys are created, one step is to identify the information need of

users at each touch point, and the resulting data flows that are required to fulfill the information needs. Likewise, the overall functionality is broken down to functionality of each touch point that results from the interface conception.

### E. 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.

### F. 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 [8] 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 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.

### G. 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.

There may be another modeling step included, though. Software may be defined in an abstract way, close to programming but abstracting from concrete programming languages and other base technologies. Software is derived from such abstract code bases by means of code generation. This helps building multi-platform software and avoiding repetitive coding tasks. This topic is revisited in Section VI.

### H. 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.

### I. Roll-out and Operations

MDSE processes are primarily concerned about the creation of a software solution. The overall software lifecycle requires the consideration of further phases. Modern development approaches take these into account, for example, by providing product increments in agile approaches and through DevOps methods.

Consequently, model-supported processes should consider project stages after software generation in due time, namely roll-out, operations, maintenance, and support. Optimally, there are explicit descriptions of the activities in those phases and for the precautions to be taken by software.

For software roll-out, for example, deployment scripts can be generated for software installation. On a higher level, and International roll-out mean orchestrating a global team of people in different roles. The orchestration needed is partly dependent on the solution design.

Other activities of a software roll-out are concerned about establishing user-acceptance, for example, by providing documentation and training. Documentation and training are, of course, dependent on the software and may partially be generated from the software models.

When it comes to operations, in particular in the presence of virtualized, eventually cloud applications that are common today, software is built in a way that allows utilizing the advantages of the operations infrastructure. For example, scalability and elasticity have to be considered in software architecture. This way, software design decisions made early in the process have have an impact on the operation of the software. For this, the models of early software design contribute to operations models.

Also, software needs to meet non-functional requirements like maintainability for the operations phase. To this end, for example a logging concept needs to be considered in the MDSE models. Remote logging is particularly important in distributed systems, for example, incorporating mobile apps.

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.

| | Data model | Progr. language | OO model | Software model |
|---|---|---|---|---|
| **Metameta Layer** | Data modeling style | Syn.+sem. definitions | Class *MetaClass* | M³L |
| **Meta Layer** | Data definition lang. | Programming lang. | MetaClasses | Metamodel |
| **Abstract Layer** | Schema | Programm | Classes | Software model |
| **Concrete Layer** | Data | Execution | Instances | Software solution |

Figure 2. Modeling layers.

## IV. MSSC Including Non-Formal Artifacts

The integration of different kinds of descriptions into the models on each modeling stage is a second respect in which we feel that existing MDSE approaches need extension.

Descriptions range from software models with defined semantics and other formalisms to unstructured media like written text and images.

### A. Formal Software Models

Existing MDSE approaches usually use formalisms that allow assigning formal semantics to models, and to apply model transformations. The formal models that are used depend on the utilization of the models, for example, UML diagrams if there is an emphasis on software architecture, as in the MDA, or Petri Nets to model behavior [9].

While these models are well-suited for MDSE approaches as they allow model transformations, they are primarily appealing to those with a formal background, for example, computer scientists. Therefore, formalisms provide a sound basis for the construction of, for example, scientific and engineering applications. But experts in other domains may be less comfortable with providing abstract models of their domain and the desired software solution.

### B. Visual Descriptions of Software

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 that cannot be expressed in the form of a given central model. For such application domains, the properties of software are designed by experts who use specific notations and tools. Digital communication like marketing and sales communication over a website is an example of such an application domain.

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.

### C. Metamodeling

The heterogeneity of models that together build the basis for MSSC raises the question of how to relate different models to each other. Our answer to that question is a common modeling base that provides a framework in which different modeling approaches can be applied.

Modeling layers that build upon each other are found in various places in computer science. A stack of four modeling layers, where the topmost layer is recursively defining itself, is found in various places. The layers range from a layer of concrete entities to a meta meta layer in these cases [10]. Figure 2 illustrates this.

The four levels of modeling are found in database models, for example, where data are on the concrete layer. Data is described by a data model or schema, that in turn is given in some data definition language. A data definition language is formulated with respect to a data modeling style, for example, the relational data model.

Also, we see for layers in programming language, where a program is an abstract concept that is instantiated in a program run on the concrete layer. The program is written in a programming language (meta layer) that builds upon general notions of syntax and semantics definitions.

In object-oriented modeling, these layers may all be expressed within one programming language. Objects are defined by classes, classes are in turn defined by metaclasses that are instances of one common metaclass.

In software engineering, there are different perspectives on a software solution and the project in which it is created, and models from different perspectives are formulated in different notations. This means that there are different languages (aka metamodels) for the co-existing perspectives that need to be integrated in one holistic modeling process. This calls for

a meta-meta-model which establishes a common ground on which metamodels are defined.

The rightmost column indicates that the modeling language used for experiments in this article, the M³L, also fits into the pattern. Being a metamodeling language, it is itself to be located at the metameta level. We chose the M³L because it allows to be applied on all levels and it does not differentiate between relationships on one level (for example, subtyping or aggregation) and relationships between level (for example, instantiation). More on this in the subsequent section.

### D. 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 3a shows the basic structure of model transformations on one stage and between stages. Figures 3b to 3g 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.

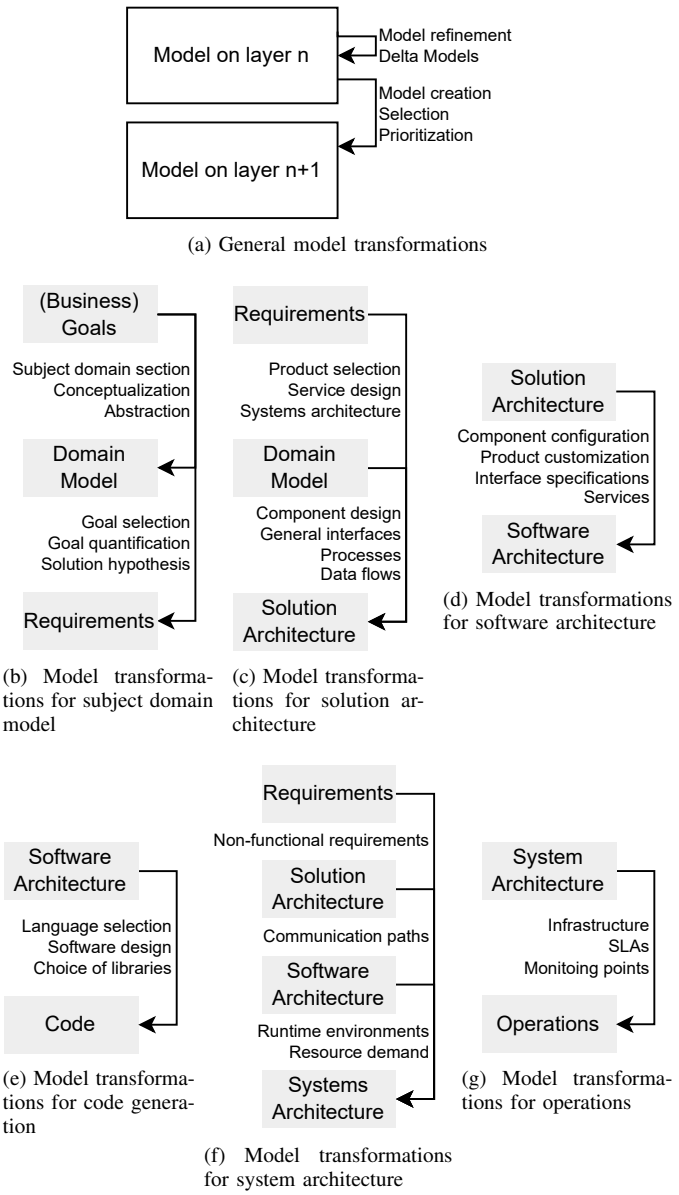
*c) Model Creation from Existing Models:* When processing from one stage to another, initial models are required for the subsequent stage that is entered. These models shall be related to the most concrete models of the preceding stage. In some cases, models can be transformed when proceeding to a subsequent 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 and Garlan [8] demand that a software architecture description refers to requirements.

## V. A Brief Introduction to the M³L

In this paper, we propose using the *Minimalistic Modeling Language*, *M³L* (pronounced "mel") [11], as the modeling framework required for MSSC.

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 demonstrate its use.




(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 3. Different kinds of model transformations.

### A. Basic Definitions

A M³L 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 }
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 }
B
```

define *B* as a specialization of *C* in the context of *A*, and without base concept in the topmost context.

A concept in a nested context is referenced as

```
B from A
```

### B. Concept Evaluation

*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* if

- *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 as found in imperative programming languages can be defined as (given *Statement*, *Boolean*, *True*, and *False*):

```
IfThenElseStatement is a Statement {
  Condition     is a Boolean
  ThenStatement is a Statement
  ElseStatement is a Statement }
IfTrueStmt is an IfThenElseStatement {
  True is the Condition
} |= ThenStatement
IfFalseStmt is an IfThenElseStatement {
  False is the Condition
} |= ElseStatement
```

A concrete program derives a conditional statement from *IfThenElseStatement*:

```
Conditional17 is a IfThenElseStatement {
 SomeBoolExpression is the Condition
 SomeStatement      is the ThenStatement
 SomeOtherStatement is the ElseStatement}
```

When evaluated, such a conditional statement will match (become a derived subconcept) of either *IfTrueStmt* or *IfFalseStmt*, depending on the concept that *SomeBoolExpression* evaluates to. from the derived base concept, the corresponding semantic rule will be inherited, making the statement evaluate to either then "then branch" or the "else branch".

Concepts are evaluated with respect to an evaluation context. Concept definitions that contribute to the evaluation of a concept are taken from that context.

### C. External Concept Representations

Concepts can be marshaled/unmarshaled 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 can be given as:

```
Java {
  IfThenElseStatement
  |- if ( Condition )
     ThenStmt
     ElseStmt . }
```

In this example, an *IfThenElseStatement* will be used to generate Java code when it is marshaled in the context of the concept *Java*.

Not that the concepts *if*, *(*, and *)* are created in this syntactic rule. Since every concept, by default, represented by its name, these concepts can be used like string literals.

## VI. 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 [12]. 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.

The three model relationships named in Section IV-D can be expressed with the M³L. This way, models are put in context.

Also, code can be generated from M³L models.

### A. Combining Models

Let *BaseModel1* and *BaseModel2* be some models of some domains whose concepts can be reused for the domain at hand. Then, for example, concepts *A* and *B* can be integrated into a new model *SomeModel* by definitions like

```
SomeModel {
  A from BaseModel1
  B from BaseModel2 }
```

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.

```
SomeModel { A { C } }
```

can be refined to

```
RefinedModel is the SomeModel {
  A is a D {
    C is an E }
  B }
```

Making the *RefinedModel* the only specialization of *SomeModel*, all references to *SomeModel* are then narrowed to *RefinedModel*.

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 of a Subsequent Stage

A model can be explicitly created as a transformation of another model using a semantic rule. For example, a model *RefinedModel* on a modeling stage *Stage1*,

```
Stage1 {
  RefinedModel { A } }
```

can be amended with a semantic rule to produce a model in a subsequent stage *Stage2*:

```
Stage2 {
  RefinedModel |= SomeDerivedModel {
    F is an A {
      G is the C }
    H {
      I } } }
```

This way, the model *SomeDerivedModel* is connected to *SomeModel* by the semantic rule, as it is its whole content. The concept *B* is not considered in the derived model.

In the example of the information system:

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

*RDMBS* from the source model *OurInfoSysConcept* is reintroduced 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.

### D. 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.

A simple example of Java code generation is shown in Section V-C.

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 ) .
    ProductSKUColumn is a Columns
    |- STOCK_KEEPING_UNIT VARCHAR(50), .
    ProductNameColumn is a Columns
    |- NAME VARCHAR(100), . }
  |- "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.

In the example, in the context of a different implementation model, syntactic rules for the generation of database access code may be defined on the concepts like *ProductDataSchema* and *ProductData*.

### E. Metaprogramming with the M³L

Instead of generating code directly by syntactic rules, code can first be modeled in an abstract way as indicated in Section V-B by the example of the if…then…else statement. This allows to consistently generate code in multiple languages.

For example, consistency of table and column names in data definition code and data access code is achieved by using the same concepts during code generation:

```
OurInfoSysDBIm
  is an OurInfoSysDataLayer, an SQL {
  ProductDataSchema is a Schema {
    PRODUCTS is a Table {
      STOCK_KEEPING_UNIT is a Colum
      NAME               is a Colum } } }
OurInfoSysDBAccess
  is an OurInfoSysDataLayer, a Java {
  Product is an Interface { ... }
  ProductImpl is a Class {
    Product is an Interface
    retrieve is a Method {
      ... SELECT * FROM
      PRODUCTS from OurInfoSysDBIm
      WHERE ... } } }
```

In this example, the table *PRODUCTS* is referenced as a concept, guaranteeing that the table and column names included in the Java code are identical to the ones used in the SQL data definition statements.

The syntactic code generation rules are inherited from the concepts defined in the code models, *SQL* and *Java* in this example.

This way, we derive a PSM and finally code by means of model transformation, where the syntactic rules are coming from language models / metamodels. This meta level aspect can be found even more explicitly in the GraSyLa [13], for example.

## VII. Conclusion

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] H.-W. Sehring, “Model-supported Software Creation: Towards Holistic Model-driven Software Engineering,” Proc. Eighteenth International Conference on Software Engineering Advances, 2023, pp. 113-118.

[2] 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.

[3] 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. 2024.3.11.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] R. Koci and V. Janousek, “Prerequisites for Simulation-Based Software Design and Deployment,” Proc. Eighteenth International Conference on Software Engineering Advances, 2023, pp. 105-109.

[10] M. Jarke, Matthias, R. Klamma, K. Lyytinen, “Metamodeling,” in Metamodeling for method engineering, M. A. Jeusfeld, M. Jarke, and J. Mylopoulos, Eds. Cambridge, MA : The MIT Press, pp. 43-88, 2009.

[11] 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.

[12] 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.

[13] V. Englebert and K. Magusiak, “The Grasyla 2.0 Language,” Edition 1.2 (Draft), University of Namur, [Online] Available from: https://staff.info.unamur.be/ven/metadone.site/documents/techreport-grasyla-version-2.1.2.pdf. 2024.3.24.