**Introduction**

Models are recognized to implement at least two roles by applying abstraction:

*-reduction* feature: the models only reflect a (relevant) selection of the original’s properties, so

as to focus on the aspects of interest; and

-*mapping* feature: the models are based on an original individual, which is taken as a prototype

of a category of individual and is abstracted and generalized to a model.

approaches based on modeling for the development of software artifacts 🡪 MDSE

The need for relying on models for software development is based on four main facts:

1. Software artifacts are becoming more and more complex and therefore they need to be discussed

at different abstraction levels

2. Software is more and more pervasive in people’s life, and the expectation is that the need for

new pieces of software or the evolution of existing ones will be continuously increasing.

3. The job market experiences a continuous shortage of software development skills with respect

to job requests.

4. Software development is not a self-standing activity: it often imposes interactions with nondevelopers

…from the standardization and tooling point of view, Business and IT modeling

and technologies are converging. This brings huge benefits to organizations, which struggle to

bridge the gap between business requirements and IT implementation.

..organizational changes that should move toward more agile approaches, combined with fostering modeling efforts

and reusable design patterns and frameworks to improve productivity, while ensuring quality and

performance.

Models as skeches,blueprints,programs.. Used in all the ways.

**MDSE principi**

MDSE can be defined as a methodology1 for applying the advantages of modeling to software

engineering activities. Generally, speaking, a methodology comprises the following aspects.

• *Concepts*: The components that build up the methodology, spanning from language artifacts

to actors, and so on.

• *Notations*: The way in which concepts are represented, i.e., the languages used in the methodology.

• *Process and rules*:The activities that lead to the production of the final product, the rules for their

coordination and control, and the assertions on desired properties (correctness, consistency,

etc.) of the products or of the process.

• *Tools*: Applications that ease the execution of activities or their coordination by covering the

production process and supporting the developer in using the notations.

*Algorithms + Data Structures = Programs*

In our new MDSE context, the simplest form of this equation would read as follows:

*Models + Transformations = Software*

Obviously, both, models and transformations need to be expressed in some *notation*, which

in MDSE we call a modeling language.

In particular, the work by Bézivin postulates that the two core relations

*representation* and *conformance* are associated to the MDSE principle, as *inheritance* and *instantiation*

were associated to the object unification principle in class-based languages.

Modeling, as opposed

to simply drawing, grants a huge set of additional advantages, including: syntactical validation,

model checking, model simulation, model transformations, model execution (either through code

generation or model interpretation), and model debugging.

ACRONIMI

*Model-Driven Development (MDD)* is a development paradigm that uses models as

the primary artifact of the development process. Usually, in MDD the implementation is

(semi)automatically generated from the models.

*Model-driven Architecture (MDA)* is the particular vision of MDD proposed by the Object

Management Group (OMG) and thus relies on the use of OMG standards.Therefore,MDA can be

regarded as a subset of MDD, where the modeling and transformation languages are standardized

by OMG.

MDE would be a superset of MDD because, as the E in MDE suggests, MDE goes beyond of the pure development activities and encompasses other model-based tasks of a complete software engineering process

Finally, we use “model-based engineering” (or “model-based development”) to refer to a softer

version of MDE. That is, the MBE process is a process in which software models play an important

role although they are not necessarily the key artifacts of the development

MDSE OVERVIEW

The *implementation* issue deals with the mapping of the models to some existing or future

running systems. Therefore, it consists of defining three core aspects.

• The modeling level: where the models are defined.

• The realization level: where the solutions are implemented through artifacts that are actually

in use within the running systems (this consists of code in case of software).

• The automation level: where the mappings from the modeling to the realization levels are put

in place.

The *conceptualization* issue is oriented to defining conceptual models for describing reality.

This can be applied at three main levels.

• The application level: where models of the applications are defined, transformation rules are

performed, and actual running components are generated.

• The application domain level: where the definition of the modeling language, transformations,

and implementation platforms for a specific domain are defined.

• The meta-level: where conceptualization of models and of transformations are defined.

the *problem space* is addressed by the *analysis* phase in the development process, while the *solution space* is addressed by the *requirement collection* phase first (defining *what* is the expected outcome), and subsequently by the *design* phase (specifying *how* to reach the objective).

The *Problem Domain* is defined as the field or area of expertise that needs to be examined to solve a problem. The focus of this definition is towards putting the attention only on the topics of interest, excluding everything else.

The *Domain Model* is the conceptual model of the problem domain, which describes the

various entities, their attributes, roles, and relationships, plus the constraints and interactions.

*Technical Spaces* represent specific working contexts for the specification, implementation, and

deployment of applications. Such working contexts typically imply a binding to specific implementation

technologies and/or specification notations.The concept of technical space is crucial for MDSE

because it enables the possibility of deciding the set of technical tools and storage formats for models,

transformations, and implementations.

During the software development process it is possible to move from one technical space

to another (as represented by the arrows in the figure). This implies the availability of appropriate

software artifacts (called *extractors*) that are able to extract knowledge from a technical space and of

others (called *injectors*) that are able to inject such knowledge in another technical space.

Models are meant to describe two main dimensions of a system: the static (or structural) part

and the dynamic (or behavioral) part. That’s why we can define the following.

• *Static models*: Focus on the static aspects of the system in terms of managed data and of

structural shape and architecture of the system.

• *Dynamic models*: Emphasize the dynamic behavior of the system by showing the execution

sequence of actions and algorithms, the collaborations among system components, and the

changes to the internal state of components and applications.

Without a doubt, multi-viewpoint modeling is one of the crucial principles of MDSE. Since

modeling notations are focused on detailing one specific perspective, typically applying a MDSE

approach to a problem may lead to building various models describing the same solution. Each

model is focused on a different perspective and may use a different notation.

Metamodeling:

exactly in the same way we define a model as an abstraction of phenomena in the real world, we can define a

*metamodel* as yet another abstraction, highlighting properties of the model itself. Metamodels basically constitute the definition of a modeling language, since they provide a way of

describing the whole class of models that can be represented by that language. While in

theory one could define infinite levels of metamodeling, it has been shown, in practice, that metametamodels

can be defined based on themselves, and therefore it usually does not make sense to

go beyond this level of abstraction. At any level where we consider the metamodeling practice, we

say that a model *conforms* to its metamodel in the way that a computer program conforms to the

grammar of the programming language in which it is written. Metamodels can be

used proficiently for:

• defining new languages for modeling or programming;

• defining new modeling languages for exchanging and storing information; and

• defining new properties or features to be associated with existing information (metadata).

Transformations are actually defined at the

metamodel level, and then applied at the model level, upon models that conform to those metamodels.

The transformation is performed between a source and a target model, but it is actually defined

upon the respective metamodels. models according to some matching rules checked upon model elements.

Such transformation rules can be defined following different approaches: the transformation

can be written manually from scratch by a developer, or can be defined as a refined specification

of an existing one. Alternatively, transformations themselves can be produced automatically out of

some higher level mapping rules between models. This technique is based on two phases:

1. defining a mapping between elements of a model to elements to another one (*model mapping*

or *model weaving*); and

2. automating the generation of the actual transformation rules through a system that receives

as input the two model definitions and the mapping between them and produces the transformations.

Industry (adoption):

Therefore, it is true that modeling has yet to cross the chasm and become mainstream, mainly

due to inflated expectations raised in the past by various MDSE initiatives, including MDA and

before that plain UML-based proposals. However, new developments in the last few years in terms

of new technologies and methods (like domain-specific languages), the widening of the scope of the

field (modeling is not only used for code-generation but also for reverse engineering, interoperability,

and software maintenance), a more pragmatic modeling approach (no need to model every detail for

every project), and the release of *de facto* standard tools (in particular the EMFopen-source modeling

framework on top of the Eclipse platform) have changed the landscape and suggest that mainstream

industrial adoption may happen sooner than later2.

A drawing tool can be considered a modeling tool only if the tool “understands” the drawings, i.e., the tool does not deal with just shapes, lines, and arrows, but understands that they represent classes, associations, and other modeling concepts. This should at least be enough to validate a model, i.e., check that the model is a correct instance of its metamodel.

MDSE USE CASES

Code-generation and model-interpretation are then two different alternative strategies to

“implement” execution tools and thus make executable models actually execute.

*Code-generation* aims at generating running code from a higher level model in order to create a

working application, very much like compilers are able to produce executable binary files from

source code. In this sense, code-generators are also sometimes referred to as *model compilers*.

This generation is usually done by means of a rule-based template engine, i.e., the code

generator consists in a set of templates with placeholders that once applied (instantiated) on the

elements in the model, produce the code.

* Problem da dodavanjem koda kod parcijalne generacije, imamo razlicite stvati u modelu I kodu, idealno bi bilo kada bi se dodati kod reflektovao na model, ili kada bi postojala rezervisana (okruzena) mjesta za dodavanje.

*Model interpretation* does not generate code from a model to create a working software application.

Instead, a generic engine is implemented, which parses and executes the model on the fly, with an

interpretation approach (exactly as interpreters do for interpreted programming languages).

Despite these benefits, this option still “scares” many MDSE users. The fact that the application

source code is not available makes you dependent from the MDSE tool vendor.

INTEROPERABILITY

Interoperability is formally defined by IEEE as “the ability of two or more systems or components

to exchange information and to use the information that has been exchanged.”

Interoperability is required in several scenarios: forward and reverse engineering (e.g., between

two consecutive tools in the development chain), tool and language evolution (to address backward

compatibility with previous versions), collaborative work (several members of the same organization

may need to work together in the same task even if they use two completely different systems to

perform the task), system integration (e.g., when, after a business acquisition, the information system

of the acquired company must communicate with the one of the parent company), and so forth.

*Model-driven interoperability (MDI)* approaches aim at defining bridges to achieve interoperability

between two or more systems by applying model-driven techniques. They work by first

making explicit the internal schema (i.e., metamodel) of each system (or each externally accessible

component of the system). Metamodels are then aligned by matching the related concepts. Finally,

model-to-model transformations exploit this matching information to export data (i.e., models)

created with the first component to data conforming to the second component’s internal schema.

The basic principle of MDI is to decompose the bridge in two main parts: a syntactic mapping and a semantic mapping.

Syntactic mapping aims at crossing different technical spaces.The idea is to use *projectors* to go from

a generally heterogeneous world (in terms of formats, techniques, etc) to a more homogeneous world,

in our case the modeling world of MDSE, and vice versa. Once in the modeling world, models can

be used as the *lingua franca* between the tools.

Semantic mapping aligns the concepts coming from the domains of both systems. This mapping

is implemented as a model-to-model transformation. The transformation re-expresses the domain

concepts of system A into a set of equivalent domain concepts understandable by system B. Metamodels

A and B can be manually generated, derived from the corresponding format description

(e.g., a XML schema when the input or output are XML documents) or automatically created if the

formats to bridge conform to a meta-format for which a bridge at the metametalevel is already available

(i.e., a generic bridge between the XML Schema language and the MDSE meta-metamodel

would allow an automatic transformation of all format descriptions, i.e., specific XML Schemas, to

metamodels).

All the actual domain knowledge is specified in the “transformation” part. The “projection”

parts (i.e., “injection” or “extraction”) only deal with technical/syntactical details, mainly concerning

the storage formats.

REVERSE ENGINEERING

This process of obtaining useful higher-level representations of legacy systems is commonly

called *reverse engineering*,

A MDRE process includes three main phases.

• *Model Discovery*. In MDRE, the idea is to switch as soon as possible from the heterogeneous

real world (with many legacy artifacts of different nature) to the homogeneous world of models,

where all artifacts are represented as a set of interrelated models.

• *Model Understanding*. MostMDRE applications will require the processing of the raw models

discovered in the previous phase in order to obtain higher-level views of the legacy systems that

facilitate their analysis, comprehension, and later regeneration. Thus, the second phase is the

*model understanding* phase where chains of model manipulation techniques are employed to

query and transform the raw models into more manageable representations

• *Model (Re)Generation*. The processed models obtained at the end of the model understanding

phase are finally used to generate and/or display the expected outcome of the reverse engineered

process

MODELING LANGUAGES

Modeling language is defined throught 3 ingrs: abstract syntax, concrete syntax and semantics

If your DSL resembles

UML too much maybe you should consider using just (a subset of ) UML and avoid inventing new

“almost–UML” notations and considering them new DSLs.

…considered at all, and the definition of a language as GPL or DSL is more a matter

of practice or subjectivity. Martin Fowler clarifies that even just the decision on whether a set of

concepts (or operations) is actually a language or just a set of operations within another language is

a matter of perspective.

Indeed, UML provides a wide set of extension mechanisms: stereotypes, constraints, tagged values, and profiles.These aspects are taken care of within the *Profile diagram*, which explicitly focuses

on the extensibility aspects: it operates at the metamodel level to show stereotypes as classes and profiles as packages. The extension relation indicates what metamodel element a given stereotype is extending.

One doubt that may arise is why one should not use normal sub-classing instead of stereotyping.

Indeed, this is an option, but the effect is quite different. Subclassing is a core operator of UML

and basically defines a special relation between two items in the same model. Stereotypes instead let

the designer define new modeling concepts, which in turn will be used for drawing models. Stereotypes

are typically used when: you want to define additional semantic constraints that cannot be

specified through standard M1-level modeling facilities; the additional semantics have significance

outside the scope of UML.

Profiles typically capture domain-specific variations and usage patterns for the

language. In a sense, profiles are domain-specific interpretation of UML, and therefore can be seen

as domain-specific languages defined by extending or restricting UML.

\*\*

Metamodeling frameworks allow the specification of metamodels using dedicated editors as

well as generating modeling editors out of the metamodels for *defining* and *validating* models. This

means that metamodels are employed: *(i) constructively* by interpreting the metamodel as a *set of*

*production rules* for building models and *(ii) analytically* by interpreting the metamodel as a *set of*

*constraints* a model has to fulfill to conform to its metamodel.

The standard metamodeling language defined by the OMG is the *Meta Object Facility*

(MOF) . Besides this OMG standard, several languages and tools for metamodeling have been

proposed in the last decade, the most prominent of which is the *Eclipse Modeling Framework* (EMF)

offering the metamodeling language *Ecore*.

Four-layered metamodeling stack

On the top layer—named M3 in OMG terms—there reside the metamodeling

languages which specify the metamodeling concepts used to define metamodels. Normally, metamodeling

languages are rather focused languages which offer a minimal set of concepts. In order to

have a finite metamodeling architecture, these languages are most often reflexively defined, meaning

that they are able to describe themselves. Thus, no additional language on top of M3 is needed to

define M3, but you can think of having again the same language on M4 as on M3. Also, from a

practical point of view, having a different language on M4 than on M3 seems to bring no further

benefits. It would be again only a structural modeling language offering similar expressibility.MOF

and Ecore are designed as reflexive languages typically used on the M3 layer.

On the M2 layer, the metamodels reside which represent modeling languages (e.g.,UML and

ER) by defining their modeling concepts. These metamodels can be instantiated to build models

on the M1 layer of the metamodeling stack. Models on the M1 layer represent systems such as a

university management system (cf. UniSystem in Figure 7.2) and typically define domain concepts

by using the language concepts defined by the metamodels on M2. On the M0 layer there are the

instances of the domain concepts which represent real-world entities, e.g., a *snapshot* of the university

management system, at a given point in time.

PROCES METAMODELOVANJA:

**Step 1: Modeling domain analysis:** According to [37], three aspects have to be considered

in the first phase of developing a modeling language: the *purpose*, *realization*, and *content* of

the language.

**Step 2: Modeling language design**: A metamodeling language is used to formalize the identified

modeling concepts by modeling the abstract syntax of the language and modeling constraints

should be formalized by using OCL.

**Step 3: Modeling language validation**

**…**In particular, *intrinsic* properties, having only primitive data values,must be distinguished from *extrinsic*

properties, which represent relationships between modeling concepts.

By introducing OCL invariants for metamodel classes, a modeling

language is more precisely defined leading to models with higher quality. This is especially

important when employing models for code-generation.Otherwise, some problems may not be detected

until the final implementation is generated where the problems may manifest as compile-time

and runtime errors, or remain undetected in the worst case.

A GCS has to define the following elements: *(i)* graphical symbols,

e.g., lines, areas, complete figures such as SVG graphics, and labels for representing textual

information, e.g., for visualizing the names of modeling elements; *(ii)* compositional rules which

define how these graphical symbols are nested and combined. For instance, a label visualizing the

name of a model element is centered within a rectangle representing the model element; *(iii)* mapping

the graphical symbols to the elements of the abstract syntax for stating which graphical symbol

should be used for which modeling concept, e.g., a specific model element type is visualized by a

rectangle.

***Approaches to GCS development.*** To specify a GCS for a modeling language defined as an

Ecore-based metamodel, the following three approaches are currently supported by different Eclipse projects.

**Mapping-centric GCS.** Protagonists from this category provide dedicated modeling languages for

describing the GCS and the mapping from the concrete syntax to the abstract syntax. This approach

is followed by the Graphical Modeling Framework16 (GMF). The language engineer has to define:

*(i)* a .gmfgraph model which defines the graphical elements used to visualize model elements (cf.

class Figure and DiagramElement in Figure 7.10); *(ii)* a .gmftool model which specifies the tool

palette,17 in particular, which icons are used to produce which model elements; and finally, *(iii)* a

.gmfmap model which actually defines the mapping between elements defined in the metamodel

and the graphical elements defined in the .gmfgraph model.

**Annotation-centric GCS.** Approaches from this category directly annotate the metamodel with

information about how the elements are visualized. This approach is supported by EuGENia18.

The EuGENia framework allows to annotate an Ecore-based metamodel with GCS information

by providing a high-level textual DSML.

**API-centric GCS**. There are also approaches allowing the implementation of graphical modeling

editors directly on the code level by providing a dedicated programming framework.

***Approaches to TCS development.*** Besides the model information, metamodels do not provide

information about the other kinds of TCS elements. For the definition of this TCS specific

information, two approaches are currently available in MDE: *(i)* having either a generic TCS or

*(ii)* a language-specific TCS.

**GenericTCS.** Such as for XML,a genericTCS may be also defined for models.This means, similar

to using object diagrams to graphically visualize models in a generic way, a textual syntax generically

applicable for all kinds of models may be applied. You can think of this textual syntax as a textual

format for specifying object diagrams. The benefit is that the metamodel is sufficient to derive a

TCS, i.e., no additional concrete syntax specification is needed.A drawback is that no tailored syntax

can be developed dealing with the specifics of a given modeling language.

**Language-specificTCS.**To eliminate the drawbacks of having solely a genericTCS, approaches for

defining specificTCSs for modeling languages have been proposed.With respect to the methodology

of the language engineering process, two approaches can be distinguished.

***Metamodel first.*** To develop the syntax of a modeling language, a metamodel first approach

may be followed. This means, first the abstract syntax is defined by the means of a metamodel. In a

second step, the textual syntax is defined based on the metamodel. Metamodel first approaches are

based on the assumption that metamodels represent the central language artifacts.

***Grammar first.*** This kind of approach follows the same goal as metamodel first approaches,

but proposes a different methodology to reach this goal. Inspired by EBNF, these approaches start

the language definition by developing the grammar defining the abstract and concrete syntax at once

as a single specification. The languages to define the grammars are also based on text production

rules as used for metamodel first approaches. In a subsequent step, the metamodel is automatically

inferred from the grammar by dedicated metamodel derivation rules.

MODEL-TO-MODEL TRANSFORMATIONS

In a general sense, a M2M transformation is a program which takes one or more models as

input to produce one or more models as output. Besides classifying model transformations based on the number of input and output models, another dimension is whether the transformation is between models specified on the basis of two different languages, referred to as *exogenous* transformations, or if transformations are defined within one language which are called *endogenous* transformations.

An example for an exogenous transformation is the typical MDA scenario where a platform independent model, e.g., a UML model,

is transformed to a platform specific model, e.g., a Java model. A well-known example for endogenous

transformations is model refactoring. Similar as for code, a model may be subject to quality

improvements which may be achieved by restructuring the models using a transformation.

During the last decade, two central execution paradigms for model transformations emerged. First, there are *out-place* transformations for *generating* the output

model from scratch. Such transformations are especially suited for exogenous

transformations. Second, there are *in-place* transformations for *rewriting* a model by creating, deleting,

and updating elements in the input model. Of course, this paradigm suits

perfectly endogenous transformations such as refactorings.

ATL is a rule-based language

which builds heavily on OCL, but provides dedicated language features for model transformations

which are missing in OCL. In a nutshell, the possibility to create model elements by providing

different kinds of rules.

source models are queried but

no changes to them are allowed. In contrast, target model elements are created, but should not be

queried directly2 during the transformation. The reason for both restrictions is that without them,

the result of queries to source and target models may differ based on the execution state of the

transformation which would be in contradiction with the nature of the declarative part of ATL.

***Anatomy of ATL transformations.*** A transformation defined in ATL is represented as a *module*

which is divided into a *header* and a *body* section. The header section of an ATL transformation

states the name of the transformation module and declares the source and target models which are

typed by their metamodels. There can be more than one input model and output model for an ATL

transformation.

The body of an ATL transformation is composed by a set of *rules* and *helpers* which are stated in

arbitrary order after the header section.

**Phase 1: Module initialization.** In the first phase, among other things, the *trace model* for storing

the *trace links* between source and target elements is initialized . Each execution of a matched rule

is stored in the trace model by creating a trace link pointing to the matched input elements and

to the created output elements. As we shall see later, the trace model is an important concept for

exogenous transformations: *(i)* to stop the execution of a transformation and *(ii)* to assign features

of the target elements based on values of the source elements.

**Phase 2:Target elements allocation.** In the second phase, the ATL transformation engine searches

for matches for the source pattern of the matched rules by finding valid configurations of source

model elements.When the matching condition of a matched rule (all input pattern elements are

bound and the filter condition is valid) is fulfilled by a configuration of source model elements, the

ATL transformation engine allocates the corresponding set of target model elements based on the

declared target pattern elements. Please note that only the elements are created, but their features

are not yet set.

**Phase 3: Target elements initialization**. In the third phase, each allocated target model element

is initialized by executing the bindings that are defined for the target pattern element.

In the bindings, invocations of the *resolveTemp* operation are quite common. This operation

allows for the reference to any of the target model elements which have been generated in

the second execution phase for a given source model element.

Graph transformations are especially useful to define in-place transformations to support,

e.g., model simulation, optimization, execution, evolution, and refactoring. Furthermore, they are

so general that also out-place transformations may be formulated with graph transformations by

representing the source and target models as well as the trace model as one integrated graph.

A *graph grammar* consists of a set of graph transformation rules and an initial graph (often

referred as *host graph*) to which the rules are applied. The core of a rule comprises a left-hand side

(LHS) graph and right-hand side (RHS) graph. The LHS expresses pre-conditions for the rule to

be applied, whereas the RHS contains the rule’s post-conditions. The actions that are going to be

carried out by the rule are implicitly defined through both sides. More precisely, the execution of a

transformation rule produces the following effects: *(i)* all elements that only reside in the LHS are

deleted; *(ii)* all elements that only exist in the RHS are added; and *(iii)* all elements that reside

in both sides are preserved.To mark that an element in the RHS is equivalent to an element in the

LHS, the elements must have the same identifier assigned.

Graph transformation rules may be applied fully automated by the graph transformation

engine as long as possible in arbitrary order. However, there are also cases which require a more

interactive execution mode.Assume there should be a refactoring applied on a specific model element.

Thus, this element has to be selected by the user and represents an important input parameter for the

transformation rule which implements this refactoring.Thus,some graph transformation approaches

allowfor pre-boundLHSelements by providing explicit input parameters which can be set by the user

before executing transformation rules.

***Advanced graph transformations techniques.*** There are several advanced techniques for specifying,

executing, and analyzing graph transformations.

**Alternative notations.** Until now, transformation rules have been defined in the abstract syntax of

the modeling language. However, an even higher readability may be reached by using the concrete

syntax of the modeling languages for defining the transformation rules.

**Rule scheduling.**

**Analysis.** Because graph transformations are a declarative approach and based on a strong theoretical

basis, there are several analysis methods for graph transformation systems available.First, in case nondeterministic

graph transformation systems are employed, there is the question if always the same

unique model is finally produced when the rules are applied in any possible order.

Transformations may be complex processes which should be modeled themselves and structured

into different steps to avoid having one monolithic transformation. Transformation chains are the

technique of choice for modeling the orchestration of different model transformations.Transformation

chains are defined with orchestration languages which allow to model in their simplest form

sequential steps of transformations.

Just as a normal model

can be created, modified, and augmented through a transformation, a transformation model can

itself be created, modified, and so on, by a so-called *Higher Order Transformation* (HOT) [60]. This

means that we can write transformations that take as input a model transformation and/or generate

a model transformation as output. For instance, we could write a HOT that automatically refactors

a set of transformations to improve their internal structure. Another example would be to add a

logging aspect to the transformation.

…two

scenarios may benefit from alternative execution strategies: *(i)* an output model already exists from a

previous transformation run for a given input model and *(ii)* only a part of the output model is needed

by a consumer, i.e., any kind of model manipulation tool such as a subsequent transformation or

modeling editor. For the former scenario, *incremental* transformations [36] may be employed.

Lazy transformations are driven by the access operations on the target model. As known

from programming languages which employ lazy evaluation for computing expressions only as far

as required in a given point in time, target models also may be produced only under request.

.. different execution modes such as *transformation*, *integration*, and *synchronization*.

The integration mode assumes to already have the source and target models given and computes

the correspondences between source and target model elements. This means, instead of generating

elements in the one model for a given match of a transformation rule in the other model, the

integration mode checks if the expected elements (that would be produced in the transformation

mode) exist in the other model. If for every match in the source model the corresponding elements

in the target can be found (and vice versa), the two models are considered to be fully integrated.

This shows quite well that bi-directional transformations may be seen as constraints that have to be

fulfilled by a pair of models.

.. if two models are not fully integrated,

the synchronization mode aims at establishing a full integration by applying some of the defined

transformation rules. Example would

be that an element is deleted in one of the models and now a match is no longer valid. Consequently,

the corresponding elements on the other side have to be deleted as well.

MODEL-TO-TEXT TRANSFORMATIONS

While in the context

of compilers, code-generation is the process of transforming source code into machine code, in

MDE, code-generation is the process of transforming models into source code. Thus, MDE codegeneration

is built on top of existing compilers for programming languages.

**How much is generated?**

**What is generated?** The motto here is: the less code to generate which is able to represent a system, the

better.

**How to generate?**

Code-generation may be described as the vertical transition from models on a higher-level of abstraction

to lower-level artifacts. The benefits of MDE are heavily based on this vertical transition.

Thus, code generators have to bridge this gap in the abstraction which may be achieved in different

ways.

idea of MDE is to abstract from technology details.

Ili protected area ili vise vremena na pripremanje modela (parametrizacija)

**BENEFITS OF M2TTRANSFORMATION LANGUAGES**

***Separated static/dynamic code.*** M2T transformation languages separate static and dynamic code

by using a *template*-based approach to develop M2T transformations. A template3 can be seen as

a kind of blueprint which defines static text elements shared by all artifacts as well as dynamic

parts which have to be filled with information specific to each particular case. (*meta-markers* for the dynamic part) Meta-markers are placeholders and have to be interpreted by a *template engine* which processes the

templates and queries additional data sources to produce the dynamic parts.

***Declarative query language.*** Within the meta-markers, code is used to access the information

stored in the models. As presented before, OCL is the choice to do this job in most M2M transformation

languages. Thus, current M2T transformation languages also allow to use OCL (or a

dialect of OCL) for specifying meta-markers.

***Reusable base functionality.*** CurrentM2Ttransformation languages come with tool support which

allow to directly read in models and to serialize text into files by just defining configuration files.

Thus, no tedious redefinition of model loading and text serializing has to be developed manually.

To deal with the complexity when implementing model-based code generators, several advanced

techniques can be utilized which are shortly described in the following.

**Apstrahovanje** – prvo napisi konk primjer pa iz toga template kreiraj

To ensure that the generated code is accepted by developers, which is especially important

when only a partial code-generation can be applied, abstracting the code-generation templates from

a concrete reference example is a proven approach.

**Reuse templates – u**nutar template-a pozovi postojeci template, dobro ide uz pricu o nasledjivanju

**Generating step-by-step.** Dividing a code-generation process into several steps by chaining

transformations is in particular useful when there is still a big gap between the model level and

the code level.

**Mastering code layout.** The code layout is determined by the template layout.

This trivial example shows quite well that one has to be careful when refactorings are applied

on the model and a partial code-generation has been already used. One way of dealing with such

kinds of refactorings may be to execute refactorings on the model level and the corresponding

refactorings on the code level, and run the code generator after the propagation of the refactorings.