Dual Deep Modeling of Business Processes

A Contribution to the Multi-Level Process Challenge

Bernd Neumayr*,[a], Christoph G. Schuetz[a], Michael Schrefl[a]

[a] Institute of Business Informatics – Data & Knowledge Engineering, Johannes Kepler University Linz

Abstract. *Multi-level modeling (MLM) facilitates conceptual modeling at multiple levels, with clabjects as basic modeling constructs that combine characteristics of metaclasses, classes and objects. Different MLM approaches differ, among others, in the meaning and structure of levels and clabjects, in the strictness or flexibility regarding cross-level relationships, and in the mechanisms for deep characterization by which clabjects at higher levels describe and constrain clabjects at multiple lower levels. The Multi-level Process Challenge provides a testbed for MLM approaches to highlight design decisions regarding these aspects. In this paper we solve the challenge using Dual Deep Modeling (DDM), a MLM approach that features dual potencies which facilitate high flexibility for cross-level relationships. With relationships with dual potencies, a single clabject can play multiple roles at different levels of instantiation, thereby DDM facilitates very compact multi-level models.*

Keywords. Deep Instantiation • Clabject • Business Process Modeling

Communicated by João Paulo A. Almeida, Thomas Kühne and Marco Montali.

1 Introduction

Multi-level modeling (MLM) facilitates conceptual modeling at multiple levels. Each level of a multi-level model specifies or affects the schema of the next lower level. Deep characterization refers to the concept that model elements at one level not only affect the next lower level but also affect levels further down. In potency-based approaches, a natural number (the potency) assigned to a model element indicates the depth of characterization, that means, broadly speaking, how many levels down that model element has a direct impact. We also say the potency indicates how many lower levels that model element covers or how many levels down that model element spans.

The clabject is the basic modeling construct of many MLM approaches and combines characteristics of object, class, and often also of metaclass. A clabject is an instance of a clabject, the latter acting as its class, at the next higher level and acts as class for its member clabjects at the next lower level. With deep characterization, a clabject not only specifies structure and behavior of its member clabjects at the next lower level, but also the structure and behavior of clabjects at levels further down.

Different MLM approaches differ, among others, in the basic modeling constructs, in the principles behind levels, and in the mechanisms for deep characterization. MLM approaches further differ with respect to the strictness or flexibility regarding cross-level relationships, i. e., regarding the question whether and how model elements at mutually different levels may be connected. The MULTI Process Challenge (Almeida et al. 2019) provides a testbed for MLM approaches to highlight their design decisions regarding these aspects.

In this paper we present our solution to the MULTI process challenge using Dual Deep Modeling (DDM), a MLM approach that features dual

* Corresponding author.
E-mail. bernd.neumayr@jku.at

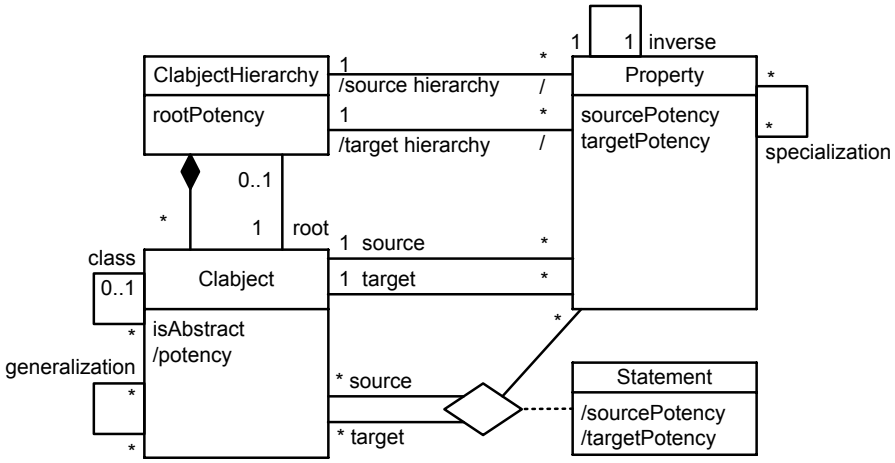

Figure 1: DDM's linguistic metamodel (Neumayr et al. 2018, p. 237)

potencies which facilitate cross-level relationships. By assigning dual potencies (consisting of a source potency and a target potency) to a property, one separately defines the depth of characterization with regard to the domain of the property (represented by its source clabject) and with regard to the range of the property (represented by its target clabject). With dual potencies, a single clabject can play multiple roles at different levels of instantiation. DDM thereby enables very compact multi-level models.

The remainder of the paper is structured as follows. Sect. 2 introduces the modeling constructs of the DDM approach. Sect. 3 presents the requirements from the challenge and makes some additions and clarifications. Sect. 4 presents and explains in detail our modeling solution to the challenge. Sect. 5 discusses how the solution satisfies the requirements. Sect. 6 makes an assessment of the modeling solution discussing choices made and potential alternative solutions. Sect. 7 discusses related work on multi-level modeling for business process management and compares our solution with existing solutions to the process challenge. Sect. 8 concludes the paper with some general observations, lessons learned and implications for future work.

2 Modeling Approach

We employ dual deep modeling (DDM), a multi-level data modeling approach based on clabjects and deep instantiation (Neumayr et al. 2018), for the representation of business process models. In the following, we briefly summarize the intuition of the main modeling concepts and rules in DDM as far as required for the solution of the multi-level process challenge.

The linguistic metamodel of DDM (Neumayr et al. 2018, p. 237) is shown in Fig. 1. Clabject hierarchies comprise multiple clabjects, with one root clabject and a root potency. The instantiation relationship between clabjects determines the hierarchical order of the clabjects. A clabject may also specialize multiple other clabjects, which are referred to as the generalizations of the specializing clabject. A property links a source clabject with a target clabject, linking also a source hierarchy with a (possibly different) target hierarchy. Each property has a source and a target potency. A property may specialize multiple other properties. Each property has an inverse property. A triple consisting of source clabject, target clabject, and property constitutes a statement having a derived source potency and a derived target potency.

In DDM, a *clabject hierarchy* consists of a number of instantiation *levels*, each associated with a *potency*. Levels are referred to by their

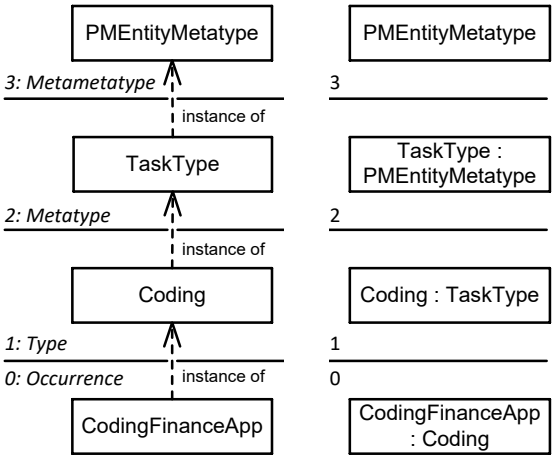

Figure 2: A DDM clabject hierarchy with explicit 'instance of' relationships and named levels (left) and its alternative and more compact representation (right).

potency and are ordered from 0 (bottom) to *n* (top or root). Levels may also have a *level name* for better understanding, e.g, level 0 in the process model hierarchy is named *Occurrence*. Every clabject belongs to one hierarchy and is situated at one level of that hierarchy; the level's potency determines the clabject's potency. A clabject's level and, consequently, potency derives from the instantiation relationships; the potency of the instance with respect to its class is reduced by 1.

Figure 2 shows the *ProcessModelHierarchy* of clabjects. At that hierarchy's top level (named *Metametatype*), the *root clabject* PMEntityMetatype specifies the hierarchy's *root potency* (3), i. e., the number of instantiation levels below the root clabject. The *clabject* TaskType instantiates the hierarchy's root clabject PMEntityMetatype with a defined potency of 3 and, therefore, TaskType is situated at the *Metatype* level with a potency of 2. The clabject Coding instantiates the clabject TaskType and, therefore, Coding is situated at the *Type* level with a potency of 1. Finally, the clabject CodingFinanceApp instantiates the clabject Coding and, therefore CodingFinanceApp is situated at the *Occurrence* level with a potency of 0.

An *abstract clabject* cannot be instantiated directly by another clabject, unless the instantiating

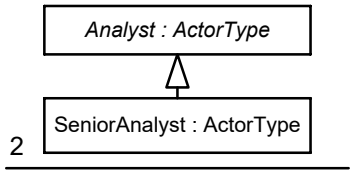

Figure 3: Abstract super-clabject and its specialization

clabject is itself abstract. Rather, an abstract clabject must be specialized before by a concrete clabject which then can be instantiated; only abstract clabjects *can* be specialized. Both the abstract clabject and its specialization are within the same hierarchy and at the same level. An abstract clabject's name is written in *slanted* font in the diagrams. For example, the abstract clabject *Analyst* in Fig. 3 is the generalization or *super-clabject* of the *concrete clabject* SeniorAnalyst. Both clabjects are at the same level (2 or *Metatype*) within the same clabject hierarchy.

The *descendants* of a clabject are the clabjects at various levels that are directly or indirectly attached to that clabject via specialization and instantiation. For example, metatype TaskType in Fig. 2 has task type Coding as descendant one level below and task occurrence CodingFinanceApp as descendant two levels below. Abstract actor type Analyst in Fig. 3 has actor type SeniorAnalyst as descendant at the same level.

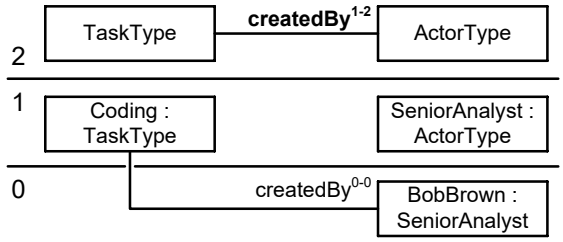

Figure 4: Deep property and property instantiation

A *deep property* is introduced between a *source* clabject and a *target* clabject with a *source potency* and a *target potency*. Dual potencies facilitate modeling of cross-level relationships. A deep property can be instantiated between descendants of the source clabject and descendants of the target clabject. The source and target potencies specify

the number of levels down the clabject hierarchy, starting from the source and target clabject, respectively, that clabjects can instantiate the property. Consider, for example, the deep property createdBy in Fig. 4 from TaskType to ActorType with source potency 1 and target potency 2. An instance of TaskType, e. g., Coding, will assign a value to the createdBy property, referring to a descendant of ActorType two levels down the hierarchy, e. g., BobBrown which is an instance of SeniorAnalyst, which instantiates ActorType. In the diagrams, a property is represented as a connector between the source clabject and the target clabject, with the property name together with the superscript dual potency displayed next to the target clabject. When a property is introduced, it is displayed in **boldface** to distinguish the property's introduction from a mere statement instantiating that property.

Any property has an implicit *inverse property*. For example, the implicit inverse property of createdBy connects ActorType as source clabject with TaskType as target clabject and has source potency 2 and target potency 1. A property together with its implicit inverse property represent the two ends of a binary association as well as a link. In DDM, everything is a clabject: atomic data types, values, object classes and objects. Likewise, every property, in combination with the implicit inverse property, represents an association over multiple instantiation levels and a link.

A *deep statement* restricts the range of a deep property further down the instantiation hierarchy for descendants of a particular clabject. For example, in Fig. 5, the produces property between TaskType and ArtifactType has a restricted range for the Coding clabject: Instances of Coding must take the assigned value for produces from the instances of Code. A deep statement not only restricts the property's range at lower levels but is also considered as a property value, e. g., the artifact type Code is considered as a value at level 1-1 of property produces. We also say that the statement instantiates the 1-1 level of the produces property, interpreted as a bi-directional link between Coding and Code: Code is a *value* of property produces of the clabject Coding and Coding is a *value* of the inverse of property produces of the clabject Code.

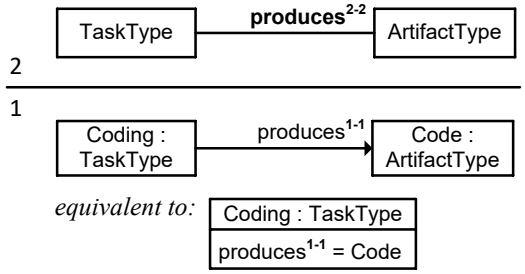

Figure 5: Deep property (top) and deep statement (bottom) restricting the property's range

A *deep statement on a deep property's inverse* restricts the range of the property's inverse for descendants of a clabject. For example, in Fig. 6, the inverse of the produces property has a restricted range for the Code clabject: Instances of Code can only be the target of produces originating from an instance of Coding.

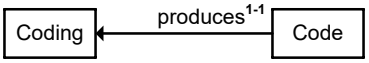

Figure 6: Deep statement on property's inverse

We introduce a shorthand notation – a line without arrowheads – for bi-directional deep statements (links) on a property and its inverse. For example, the produces link between Coding and Code (Fig. 7) represents two statements at once: for descendants of Coding, a restriction of the range of produces to descendants of Code, and for descendants of Code, a restriction of the range of the inverse of produces to descendants of Coding.

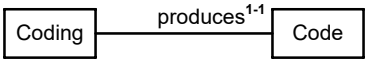

Figure 7: Link

A *statement ensemble* is a set of deep statements that connect a source clabject with a set of clabjects as target of the same property at the same target potency. For example, in Fig. 8, a statement

ensemble for the clabject SoftwareDevelopment over the task property restricts the range of the task property for instances of SoftwareDevelopment to the union of instances of Design and instances of Coding.

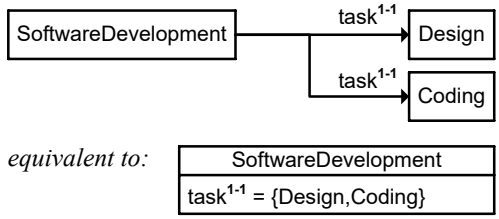

Figure 8: Statement ensemble

DDM also allows for the specialization of properties. For example, in Fig. 9, the initialTask property refines task. We refer to initialTask as the *sub-property* of task and, in turn, we refer to task as the *super-property* of initialTask. The sub-property derives the target and source potencies from the super-property. A sub-property's values are propagated upwards to the super-property, e. g., the extension of the task property comprises the initialTask values. The range refinements on a super-property via deep statements are inherited by the sub-properties.

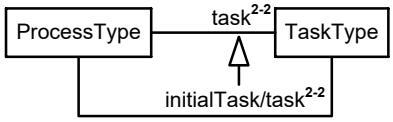

Figure 9: Property specialization

Deep cardinality constraints consist of source clabject, deep property, target clabject, multiplicity, and the to-be constrained level of the deep property indicated by dual potencies. Depending on the multiplicity, we talk of mandatory constraint (1..*), functional constraint (0..1), or mandatory and functional constraint (1..1). For example, in Fig. 10, a mandatory constraint over level 1-1 of the in property between GatewayType and TaskType requires each instance of GatewayType to be related to at least one instance of TaskType via the in property.

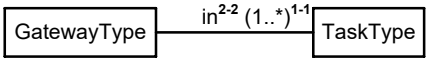

Figure 10: Deep cardinality constraints

Other multiplicities, e. g., (2..5) to indicate a minimum cardinality of 2 and a maximum cardinality of 5, are also possible but were not considered in DDM's original formalization, yet can easily be added. Since all cardinality constraints in our solution are value cardinality constraints, we omit the explicit distinction between value and range cardinality constraints provided in the original DDM notation.

3 Case Analysis

In this section, we present and analyze the case description. In order to make the paper self-contained we include the rules and requirements from the case description (Almeida et al. 2019) but leave out the examples from the insurance domain. We make additional assumption where necessary. The rules and requirements taken verbatim from the challenge description are set in sans serif font.

P1) A *process type* is defined by the composition of one or more *task types* and their relations.

P2) Ordering constraints between *task types* of a *process type* are established through *gateways*, which may be *sequencing*, *and-split*, *or-split*, *and-join* and *or-join*.

We assume that a split gateway has one incoming task and at least two outgoing tasks. Similarly, we assume that a join gateway has at least two incoming tasks and one outgoing task. A sequence has one incoming and one outgoing tasks.

P3) A *process type* has one *initial task type* (with which all its executions begin), and one or more *final task types* (with which all its executions end).

We assume that a process occurrence has exactly one final task occurrence which

must be an occurrence of one of the final task types.

P4) Each *task type* is *created by* an *actor*, who will not necessarily perform it.

P5) For each task type, one may stipulate a set of *actor types* whose instances are the only ones that may *perform* instances of that *task type*.

P6) A *task type* may alternatively be assigned to a particular set of *actors* who are authorized.

P7) For each *task type* one may stipulate the *artifact types* which are *used* and *produced*.

P8) *Task types* have an *expected duration* (which is not necessarily respected in particular occurrences).

P9) *Critical task types* are those whose instances are *critical tasks*; each of the latter must be *performed* by a *senior actor* and the artifacts they produce must be associated with a *validation task*.

P10) Each *process type* may be enacted multiple times.

We assume that every process is an instance of (i. e., enacts) exactly one process type.

P11) Each *process* comprises one or more tasks.

P12) Each *task* has a *begin date* and an *end date*.

P13) *Tasks* are associated with *artifacts used* or *produced*, along with *performing actors*.

P14) Every *artifact used* or *produced* in a *task* must instantiate one of the *artifact types* stipulated for the *task type*.

P15) An *actor* may have more than one *actor type*.

As P15 seems to check whether an approach supports multiple classification and DDM does not, we initially assume that every actor has a single actor type. But, in our opinion, the specific modeling requirement is best captured by using roles and role types, which can be incorporated with DDM at the meta type level, as discussed in Sect. 5.

P16) An *artifact* may have more than one *artifact type*.

As for actors we also assume that every artifact has exactly one artifact type and refer to Sect. 5 for a discussion of workarounds.

P17) An *actor* who performs a *task* must be authorized for that task. Typically, a class of actors is automatically authorized for certain classes of tasks.

P18) *Actor types* may *specialize* other *actor types* in which case all the rules that apply to instances of the specialized *actor type* must apply to instances of the specializing *actor type*. For example, if a *manager* is allowed to *perform tasks* of a certain *task type*, so is a *senior manager*.

P19) All modeling elements, at all levels, must have a *last updated* value of type *time stamp*. This feature should be defined as few times as possible, ideally only once. Respective definitions are exempt from the requirement to have a *last updated* value.

Using the generic process modeling language, the *Acme software development process* shall be represented. The initial task of that process is *Requirements Analysis* followed by *Design* and *Test Case Design* in parallel. *Coding* and *Test Design Review* follow the completion of *Design* and *Test Case Design*, respectively. The final task,

which follows the completion of both *Coding* and *Test Design Review*, is *Testing*.

S1) A *requirements analysis* is performed by an *analyst* and *produces* a *requirements specification*.

S2) A *test case design* produces *test cases*.

S3) An occurrence of *coding* is *performed by* a *developer* and *produces code*. It must furthermore reference one or more *programming languages* employed.

S4) *Code* must reference the *programming language(s)* in which it was written.

S5) *Coding in COBOL* always produces *COBOL code*.

S6) All *COBOL code* is written in *COBOL*.

S7) *Ann Smith* is a *developer*; she is the only one allowed to perform *coding in COBOL*.

S8) *Testing* is *performed by* a *tester* and *produces* a *test report*.

S9) Each *tested artifact* must be associated to its *test report*.

S10) *Software engineering artifacts* have a responsible *actor* and a *version number*. This applies to *requirements specification*, *code*, *test case*, *test report*, but also to any future types of *software engineering artifacts*.

S11) a) *Bob Brown* is an *analyst* and *tester*.
b) He has *created* all *task types* in this *software development process*.

We have split rule (S11) into two rules, (S11-a) and (S11-b), so that we can discuss their fulfillment in separation. As stated with rule (P15), the DDM approach does not support multiple classification and, therefore, rule (S11-a) will not be fulfilled. We discuss workarounds in Sect. 5 and, in our main solution, we treat *Bob Brown* as a senior analyst.

S12) The *expected duration* of *testing* is *9 days*.

S13) *Designing test cases* is a *critical task* which must be performed by a *senior analyst*. *Test cases* must be validated by a *test design review*.

Additional Details

To better illustrate our solution to the challenge, we supplement the case description with additional details at the occurrence level. In our solution to the challenge (Figs. 11–15), we clearly distinguish between the model elements that relate to the challenge (shown with bold black lines) and those model elements that relate only to the additional details (shown with thin gray lines). The latter can be ignored when comparing our solution to other solutions of the challenge. The additional details are the following:

A1) *Lisa Loud* is an analyst but not a senior analyst.

A2) Occurrences of *Design* may only be performed by *Lisa Loud* or by *Bob Brown*.

A3) The development of a new finance application (*DevelopFinanceApp*) is an occurrence of the Acme software development process with task *AnalyzeFinaceAppReqs*, an occurrence of *ReqAnalysis*, followed by *DesignFinanceApp* and *DesignTestsForFinanceApp*, occurrences of *Design* and *TestCaseDesign*, respectively. Both are followed, respectively, by *CodingFinanceApp*, an occurrence of *Coding*, and *ReviewFinanceAppTestDesign*, an occurrence of *TestDesignReview*. The final task after completion of those two tasks is then *TestFinanceApp*, an occurrence of *Testing*.

A4) The coding of the new finance application employs COBOL, is performed by *Ann Smith* and produces *FinanceAppCode*.

A5) The testing of the code is performed by *Peter Parker* and produces a test report having *FinanceAppCode* as tested artifact.

A6) The design of the new finance app is performed by *Lisa Loud*.

4 Multi-level Modeling Solution

In this section, we present a multi-level model as a solution to the challenge. We present our solution in three steps, each step focusing on a fragment of the overall multi-level model, covering at each step all levels, namely the metatype level, the type level, and the occurrence level. In the first step, we give an overview of our solution and present its basic ideas along Fig. 11. We will discuss modeling limitations related to abstract clabjects along Fig. 12. In the second step we focus on processes specified by gateways and tasks at the metatype level and the type level (Fig. 13), and at the occurrence level (Fig. 14). In the third step we focus on actors and artifacts and their relationships with tasks as shown in Fig. 15.

4.1 Overview and Basic Ideas

We will now present the basic ideas of our solution by considering only a fragment of the solution (see Fig. 11). At the type level, we consider only two task types, namely *Design* and *Coding*. At the metatype level, we model only one gateway metatype, namely SequenceType. As a further simplification, for now, we do not model cardinality constraints.

DDM facilitates the modeling of different clabject hierarchies with different numbers of instantiation levels and with different level names. But for simplicity's sake we opted for solving the process challenge with a single clabject hierarchy.

4.1.1 Clabjects and Instantiation Levels

The multi-level process model hierarchy is rooted in the PMEntityMetatype (short for 'process model entity metatype') clabject, which has potency 3, meaning that it has three instantiation levels, namely the Metatype, the Type, and the Occurrence levels. Clabjects in DDM are understood as multi-faceted constructs. Clabject PMEntityMetatype acts as class of process model entity metatypes at level 2, as class (and metaclass) of process model entity types at level 1, and as class (and metaclass and metametaclass) of process model entity occurrences at level 0.

With regard to these different class facets, clabject PMEntityMetatype has members at level Occurrence, such as CodingFinanceApp, members at level Type, such as Coding, and members at level Metatype, such as TaskType. Clabject PMEntityMetatype itself is at level Metametatype and is the only instance of its singleton class facet.

In turn, clabject TaskType, with potency 2 is an instance of PMEntityMetatype, has two instantiation levels, namely Type and Occurrence, and acts as class of task types and as class (and metaclass) of task occurrences. Clabject Coding, an instance of TaskType, has one instantiation level, namely Occurrence, and acts as class of coding occurrences, with CodingFinanceApp as instance.

Clabject ProcessType with potency 2 is another instance of PMEntityMetatype and acts as class of process types and as class (and metaclass) of process occurrences.

Note that we call all model entities clabjects since in DDM every model entity, even one with potency 0, has at least one class facet; a clabject always represents a class of which it is the only member (note that this is different for abstract clabjects).

4.1.2 Abstract clabjects

DDM adheres to the abstract superclass rule (Hürsch 1994) and translates it into the abstract super-clabject rule. In DDM a generalized clabject (or super clabject) is always treated as an abstract class, i. e., cannot be instantiated directly, and as an abstract object, i. e., it is not treated as a proper object with properties describing itself.

In our solution, GatewayType is an abstract metatype specialized by concrete metatype SequenceType. Clabject GatewayType also acts as

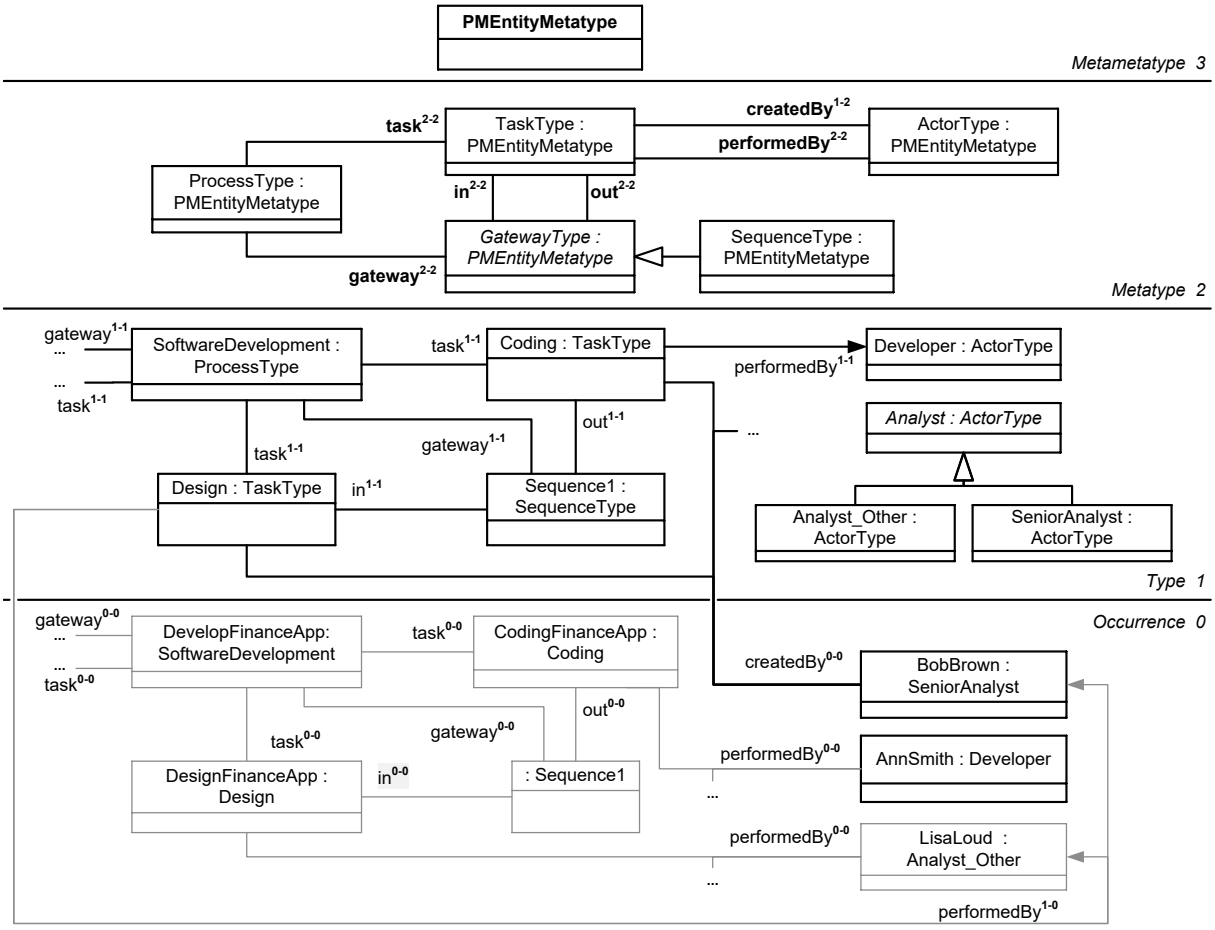

Figure 11: A fragment of the solution. Connectors ending in '…' indicate additional statements which will be introduced and described later. Model elements with thin gray lines do not refer to the challenge but to the additional details introduced in Sect. 3 for illustrative purposes.

class of gateway metatypes, with its concrete specializations such as SequenceType as members, as class of gateway types, and as class of gateway occurrences. There are no gateway types that are direct instances of metatype GatewayType but only direct instances of concrete specializations of GatewayType, such as of SequenceType. Furthermore, GatewayType cannot have its own values, e. g., specifying a 'last updated' value, its property values are propagated to its instances. Not allowing abstract clabjects to have attributes describing themselves was a compromise in the design of DDM between simplicity and expressibility where we chose simplicity.

In order to model a last updated time value for all modeling elements at all levels, clabject PMEntityMetatype (see Fig. 12) introduces four properties with different source potencies, one for each level. All four 'last updated' properties are introduced with target clabject Timestamp and target potency 1. Property lastUpdatedMMT is introduced with source potency 0, meaning that it is to be instantiated by PMEntityMetatype itself. Property lastUpdatedMT is introduced with source potency 1, meaning that it is to be instantiated one instantiation level below PMEntityMetatype, that is at the metatype level. It is instantiated by metatypes TaskType and SequenceType. Abstract metatype GatewayType does not have its own property values and hence does not have a 'last updated' value. If GatewayType specified a value for lastUpdatedMT it would be shared by (i. e., propagated to) all concrete specializations of GatewayType. Property lastUpdatedT is introduced with source potency 2, meaning that it is to be instantiated by instances of instances of PMEntityMetatype, hence by product model entity types such as Coding or Sequence1. Property lastUpdatedO is introduced with source potency 3, meaning that it is to be instantiated by instances of instances of instance of PMEntityMetatype, or three instantiation levels below PMEntityMetatype, hence by product model entity occurrences such as CodingFinanceApp or by the unnamed instance of Sequence1.

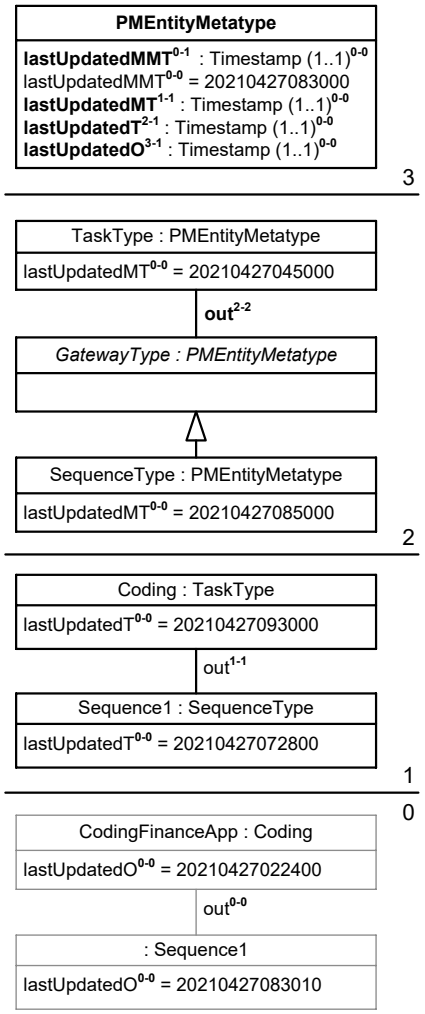

Figure 12: Modeling lastUpdate properties at the metametatype, metatype, type, and occurrence level. Model elements with thin gray lines do not refer to the challenge but to the additional details introduced in Sect. 3 for illustrative purposes.

Dual potencies need to be asserted only when declaring properties. Dual potencies of statements can be derived, they are shown in the diagrams for understandability. A statement's source potency is derived as the property's source potency reduced by the number of instantiation steps between the property's source clabject and the statement's source clabject. A statement's target potency is derived as the property's target potency reduced by the number of instantiation steps between the property's target clabject and the statement's target clabject.

4.1.3 Relationships

Process types consist of task types and gateway types. Process occurrences consist of task occurrences and gateway occurrences. In our solution this is modeled as follows. Property task between source clabject ProcessType and target clabject TaskType has source potency 2 and target potency 2, i. e., dual potencies 2-2. Statements with property task with source potency 1 and target potency 1 connect process types, i. e., members of clabject ProcessType, with task types, i. e., members of clabject TaskType; for example, process type SoftwareDevelopment is connected to task type Coding. Statements with property task and source potency 0 and target potency 0 connect process occurrences, i. e., members of members of clabject ProcessType, with task occurrences, i. e., members of members of clabject TaskType; for example, process occurrence DevelopFinanceApp is connected to task occurrence CodingFinanceApp. Property gateway between source clabject ProcessType and target clabject GatewayType is interpreted analogously.

The order of task types within a process type is modeled by gateway types and their incoming task types, connected via property in, and their outgoing task types, connected via property out. The order of task occurrences has to adhere to the order specified at the type level. This is accomplished by gateway occurrences, i. e., members of gateway types, which connect incoming and outgoing task occurrences via properties in and out, the range of which is specified by the gateway type. This is modeled by properties in and out between source clabject GatewayType and target clabject TaskType with dual potencies 2-2. Statements with properties in and out with potencies 1-1 connect a gateway type with its *in*coming task type(s) and its *out*going task type(s), respectively. For example, gateway type Sequence1 connects Design with Coding. By this, the range of property in for occurrences of Sequence1 is restricted to occurrences of Design and for property out it is restricted to occurrences of Coding. Statements with properties in and out with potencies 0-0 connect a gateway occurrence with its incoming task occurrence(s) and its outgoing task occurrence(s), respectively. For example, unnamed gateway occurrence :Sequence1 connects task occurrence DesignFinanceApp with task occurrence CodingFinanceApp.

Property createdBy connecting source clabject TaskType with target clabject ActorType has source potency 1 and target potency 2. The ultimate instances of createdBy, i. e., statements with potencies 0-0, connect task types with individual actors. For example, task types Coding and Design were created by BobBrown.

Property performedBy connecting source clabject TaskType with target clabject ActorType has source potency 2 and target potency 2. An ultimate instance of performedBy connects a task occurrence with the individual actor that performed it. For example, CodingFinanceApp was performed by AnnSmith.

Statements at intermediate levels also represent range restrictions for lower levels. For example, the performedBy statement between Coding and Developer restricts the range of performedBy for occurrences of Coding to actors of type Developers. The arrowhead indicates the direction of the range restriction, a developer may perform other tasks as well, but a Coding task can only be performed by a developer.

The statement ensemble with source clabject Design, property performedBy and dual potencies 1-0, restricts for occurrences of Design the range of performedBy to BobBrown and LisaLoud.

4.2 Processes, Tasks, and Gateways

We will now discuss in detail the modeling of processes composed of tasks and ordered by gateways at the metatype level, the type level, and at the occurrence level.

4.2.1 Metatype level

In Fig. 13, the inverse of property gateway between ProcessType and GatewayType is modeled with a mandatory and functional constraint for dual potencies 0-0, meaning that every gateway occurrence belongs to exactly one process occurrence, and a mandatory and functional constraint for dual potencies 1-1, meaning that every gateway type belongs to exactly one process type. The inverse of property task between ProcessType and TaskType comes with similar cardinality constraints.

Property task comes with mandatory constraints for potencies 1-1 and for potencies 0-0, meaning that every process type has at least one task type, and every process occurrence has at least one task occurrence. Properties initialTask and finalTask are modeled as sub-properties of task. Property initialTask comes with functional and mandatory constraints for potencies 1-1 and potencies 0-0, meaning that every process type has exactly one initial task type, and every process occurrence has exactly one initial task occurrence. Property finalTask comes with a mandatory constraint for potencies 1-1 and a mandatory and functional constraint for potencies 0-0, meaning that every process type has at least one final task type, and every process occurrence has exactly one final task occurrence. Cardinality constraints for properties in and out between GatewayType and TaskType are refined with regard to concrete gateway metatypes.

Properties in and out are introduced with mandatory constraints for potencies 0-0, meaning that every gateway occurrence has at least one incoming task occurrence and at least one outgoing task occurrence, and mandatory constraints for potencies 1-1, meaning that every gateway type has at least one incoming task type and at least one outgoing task type. The inverses of properties in and out come with functional constraints for potencies 0-0 and 1-1, meaning that at the occurrence level and at the type level a task can be incoming task of at most one gateway and outgoing task of at most one gateway.

The concrete gateway metatypes, i. e., SequenceType, AndSplitType, AndJoinType, OrSplitType, and OrJoinType, come with refined cardinality constraints. For example, AndSplitType adds functional constraints for property in at the type level and at the instance level; and for property out it adds a minimum cardinality of 2 both at the type and the instance level. Note that in the published version of DDM one can only specify functional and mandatory constraints but not an arbitrary minimum cardinality as used here, yet this can be easily added.

Clabject TaskType, as source clabject, introduces property expectedDuration with potencies 1-1 and Number as target, together with a mandatory and functional constraint for potencies 0-0, meaning that every task type comes with exactly one expected duration given as a number. It further introduces properties beginDate and endDate with source potency 2, target potency 1 and target Date, together with a mandatory and functional constraint for potencies 0-0, meaning that every task occurrence comes with exactly one beginDate and exactly one endDate.

4.2.2 Type Level

Looking at the type level, the ACME SoftwareDevelopment process type is connected to its task types and gateway types via statements with properties initialTask, task, finalTask, and gateway, all with dual potencies 1-1.

Sets of statements with the same source clabject, same property and same dual potencies form a statement ensemble, and restrict, for members of the source clabject, the range of that property to members of target clabjects. For example, the set of task statements connecting SoftwareDevelopment to task types ReqAnalysis, Design, Coding, TestCaseDesign, TestDesignReview, and Testing, forms a statement ensemble, restricting, for occurrences of SoftwareDevelopment, the range of

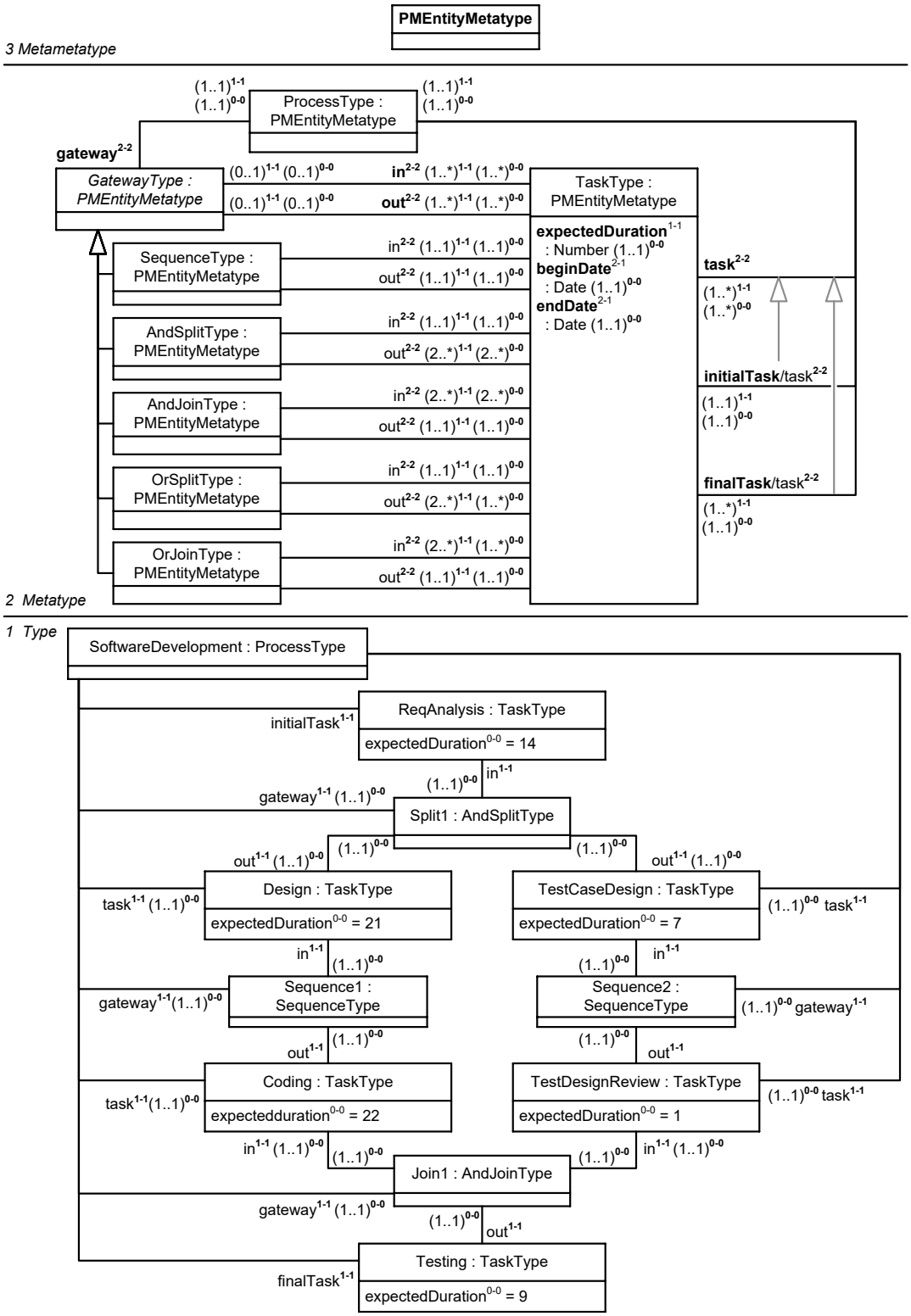

Figure 13: Modeling processes, tasks, and gateways at the metatype level and at the type level.

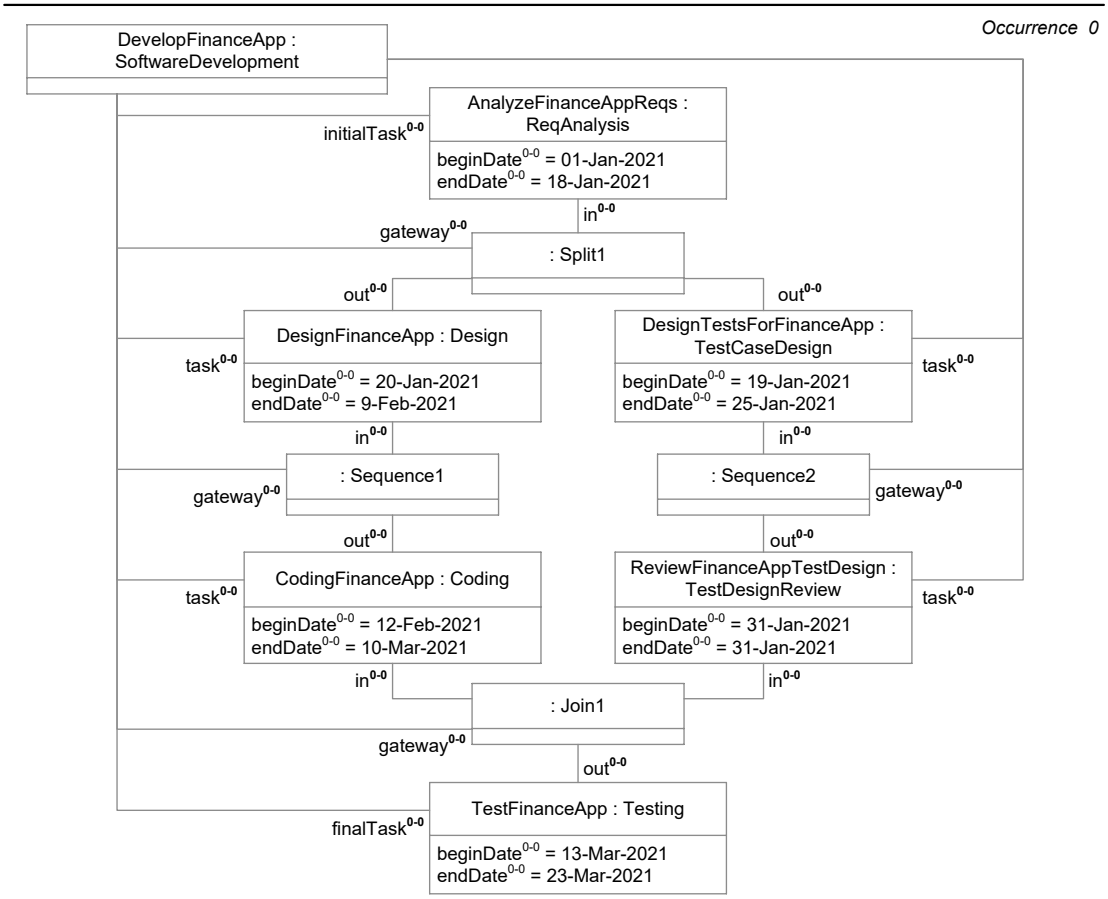

Figure 14: Modeling an occurrence of the ACME software development process. This is not part of our solution to the challenge but refers to the additional details introduced in Sect. 3 for illustrative purposes.

property task to occurrences of these task types. Likewise, the set of gateway statements forms a statement ensemble restricting the range of gateway to occurrences of gateway types Split1, Sequence1, Sequence2, and Join1. In other words, an occurrence of SoftwareDevelopment may only have tasks and gateways that are occurrences of task types and gateway types associated with the SoftwareDevelopment process type. A mandatory and functional constraint for potencies 0-0 ensures that an occurrence of SoftwareDevelopment has exactly one occurrence of each of these task and gateway types.

The cardinality constraints model successfully finished process, task, and gateway occurrences. For example, the cardinality constraints specified with gateway type Split1 should be checked once an occurrence of that split is completed, or, more exactly, once the incoming and outgoing task occurrences are available in the model. A completed occurrence of Split1 must be linked to exactly one occurrence of SoftwareDevelopment, via in to exactly one occurrence of ReqAnalysis, and via out to exactly one occurrence of Design and to exactly one occurrence of TestCaseDesign.

The in and out statements connecting task types and gateway types restrict the range of in and out statements for occurrences of the connected types. For example, gateway type Split1, an instance of gateway metatype AndSplitType, has incoming task type ReqAnalysis, and outgoing task types Design and TestCaseDesign. Occurrences of Split1 can have only occurrences of ReqAnalysis as incoming tasks and can only have occurrences of Design or TestCaseDesign as outgoing tasks.

Cardinality constraints for properties in and out for potency 0-0, specified at the metatype level and additionally at the type level, ensure that each of the in and out statements at the type level is instantiated exactly once at the occurrence level. For example, every occurrence of Split1 must have exactly one occurrence of ReqAnalysis as incoming task occurrence (the respective functional and mandatory is specified at the metatype level for gateway metatype AndSplitType) and every occurrence of ReqAnalysis must be the incoming task of a Split1 occurrence (specified at the type level). Furthermore, cardinality constraints ensure that there is a one-to-one relationship between occurrences of Split1 and Design as well as between occurrences of Split1 and TestCaseDesign.

As specified at the metatype level, every task type has a value for property expectedDuration. For example, TestCaseDesign has an expected duration of 7.

4.2.3 The Occurrence Level

A model of the DevelopFinanceApp, which is an occurrence of the ACME SoftwareDevelopment process type, is shown in Fig. 14. It has exactly one occurrence for each task and gateway type specified with the process type. Every task occurrence specifies a value for properties beginDate and endDate. All statements adhere to range restrictions and cardinality constraints introduced at the type and metatype levels.

The implementation of DDM in F-Logic (Neumayr et al. 2018) can be used to check these range restrictions and constraints at all levels.

4.3 Actors, Artifacts, and Tasks

In this subsection we present the part of the solution concerned with artifacts and actors at the metatype, type, and occurrence level.

This part of our modeling solution makes heavy use of clabject generalization and, hence, of abstract clabjects. Clabjects SoftwareEngineeringArtifact, CriticalArtifact, ValidationTask, CriticalTask, SeniorActor, and Analyst represent abstract process model entity types and COBOLCode and CodingInCOBOL represent abstract occurrences. These abstract clabjects are not treated as objects in their own right but only as generalization of concrete clabjects; they cannot have its own property values which would describe the abstract clabject as such and are not counted when checking cardinality constraints. Abstract clabjects are used to make statements and to introduce properties and constraints that are valid for their concrete sub-clabjects.

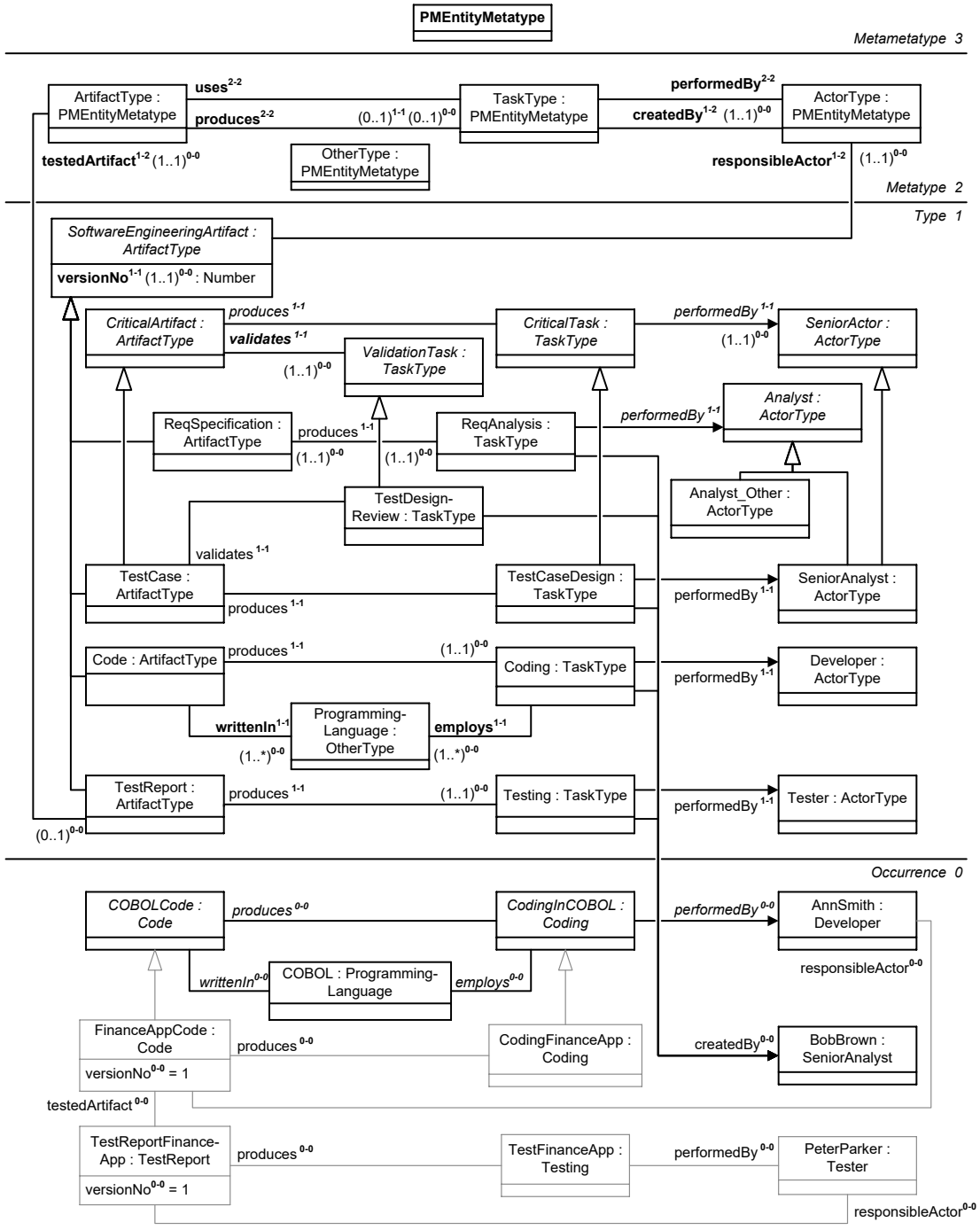

Figure 15: Modeling tasks, actors, and artifacts at the metatype level, type level and the occurrence level. Model elements with thin gray lines do not refer to the challenge but to the additional details introduced in Sect. 3 for illustrative purposes.

4.3.1 Artifacts

Metatype ArtifactType has artifact types as members which in turn have artifact occurrences (also called 'individual artifacts') as members. ArtifactType is instantiated by concrete artifact types ReqSpecification, TestCase, Code, and TestReport which are generalized by abstract artifact type SoftwareEngineeringArtifact.

FinanceAppCode is an individual Code artifact and TestReportFinanceApp is an individual artifact of type TestReport. For the other artifact types we have not modeled any occurrences.

COBOLCode is an abstract individual artifact. It is modeled at the occurrence level as a generalization of all individual Code artifacts written in COBOL. Currently, FinanceAppCode is the only concrete individual COBOLCode artifact.

Abstract artifact type SoftwareEngineeringArtifact generalizes artifact types ReqSpecification, TestCase, Code, and TestReport (further artifact types may be added later). It introduces property versionNo with potencies 1-1 and functional and mandatory constraints for potencies 0-0, specifying that every individual software engineering artifact has such a version number. For example FinanceAppCode and TestReportFinanceApp both have versionNo 1.

4.3.2 Tasks

Clabject TaskType is instantiated by task types ReqAnalysis, TestDesignReview, TestCaseDesign, Coding, and Testing. ValidationTask is an abstract task type with currently only one concrete validation task type, namely TestDesignReview. CriticalTask is another abstract task type; currently there is only one concrete critical task type, namely TestCaseDesign.

Abstract clabject CodingInCobol represents an abstract occurrence of Coding generalizing occurrences of Coding that employ COBOL. Concrete clabject CodingFinanceApp represents a concrete occurrence of CodingInCOBOL.

4.3.3 Actors

The ActorType metatype has actor types as members which in turn have actor occurrences, also referred to as individual actors, as members. Metatype ActorType is instantiated by actor types SeniorAnalyst, Analyst_Other, Developer, and Tester. Clabjects SeniorActor and Analyst represent abstract actor types. Because DDM adheres to the abstract superclass rule, Analyst has to be modeled as an abstract clabject, yet to be able to model analysts that are not senior analysts we introduce clabject Analyst_Other. Analyst_Other and SeniorAnalyst are concrete Analyst types with SeniorAnalyst being also a SeniorActor type.

AnnSmith is a developer, BobBrown a senior analyst, and PeterParker a tester.

4.3.4 Performed By

Property performedBy with dual potencies 2-2 models that task occurrences are performed by individual actors.

Statements (or statement ensembles) with property performedBy and dual potencies 1-1 at the type level specify that only actors of the given actor types are authorized to perform occurrences of the given task type. For example, the statement «TestCaseDesign is performedBy SeniorAnalyst»with potencies 1-1 models that only senior analysts are authorized to perform occurrences of test case design. The arrowhead indicates that the statement only restricts the range in one direction, and senior analysts may be authorized to perform also occurrences of other task types, such as requirements analysis.

The abstract statement «ReqAnalysis is performedBy Analyst» models that senior analysts as well as analysts that are not senior analysts (represented by class Analyst_Other) are authorized to perform occurrences of requirement analysis.

The abstract statement «CriticalTask is performedBy SeniorActor» together with a mandatory and functional constraint for potencies 0-0 models that every occurrence of a critical task type is performed by exactly one senior actor.

Statements (and statement ensembles) with property performedBy and dual potencies 1-0 connect task types with individual actors to model that only these actors are authorized to perform occurrences of the respective task type. For example,

see Fig. 11, Design is connected by performedBy statements to BobBrown and LisaLoud, to model that only these two actors are authorized to perform occurrences of Design.

Statements with abstract source clabjects are propagated to their concrete specializations. Abstract task occurrence CodingInCOBOL is connected by a performedBy statement to developer AnnSmith. The system derives the statement «CodingFinanceApp is performedBy AnnSmith».

4.3.5 Created By

Property createdBy introduced with metatype TaskType as source clabject, target clabject ActorType, potencies 1-2 and functional and mandatory constraints for potencies 0-0, models that every task type is created by exactly one individual actor. For example, the Coding task type is createdBy by individual actor BobBrown. Similarly, task types ReqAnalysis, TestDesignReview, TestCaseDesign, and Testing are createdBy BobBrown. Each of these level-crossing relationships with potencies 0-0 links a clabject at the type level to a clabject at the occurrence level.

4.3.6 Responsible Actor

Abstract artifact type SoftwareEngineeringArtifact as source clabject introduces property responsibleActor with target clabject ActorType, potencies 1-2 and a mandatory and functional constraint for potencies 0-0, meaning that every individual software engineering artifact is linked to exactly one individual responsible actor.

This level-crossing relationship associates a clabject at the type level with a clabject at the metatype level, yet its ultimate instances (the statements with property responsibleActor and potencies 0-0) connect clabjects at the same level, namely at the occurrence level. For example, artifact occurrence TestReportFinanceApp has PeterParker as responsibleActor, and FinanceAppCode has AnnSmith as responsibleActor.

4.3.7 Tested Artifacts

Artifact type TestReport, as source clabject, introduces a property testedArtifact with target clabject ArtifactType at the metatype level. The potencies 1-2 indicate that the ultimate instances link TestReport occurrences to individual artifacts, i. e., members of members of ArtifactType, with the latter in the role of tested artifacts. The cardinality constraints specify that every test report is linked to exactly one tested artifact occurrence and every artifact occurrence is linked to at most one test report occurrence.

4.3.8 Used and Produced Artifacts

Property uses introduced with metatype TaskType, target clabject ArtifactType and potencies 2-2, models that task occurrences can use artifact occurrences (note, this property is not further used in the solution).

Property produces introduced with source clabject TaskType, target clabject ArtifactType and potencies 2-2, models that artifact occurrences are produced by task occurrences. Statements with property produces and potencies 1-1 connect artifact types to task types to specify which tasks can produce which artifacts.

4.3.9 Programming Language

Clabject ProgrammingLanguage is an instance of metatype OtherType. The latter is introduced at the metatype level, to flexibly introduce additional clabjects at the type level.

Artifact type Code introduces property writtenIn and task type Coding introduces property employs both with potencies 1-1 and with ProgrammingLanguage as target clabject. The mandatory constraints indicate that each occurrence of Code and each occurrence of Coding is linked via property writtenIn or via property employs, respectively, to at least one programming language.

At the occurrence level, programming language COBOL is linked to abstract clabject COBOLCode and to abstract clabject CodingInCOBOL representing shared statements propagated to every COBOLCode occurrence as well as every CodingInCOBOL occurrence, respectively, so that they are linked to COBOL without the need to explicitly represent this link.

5 Satisfaction of Requirements

In this section, we evaluate the presented solution with respect to the case's requirements presented in Sect. 3. An overview is given in Tab. 1.

Requirement (P1) is fulfilled by clabjects ProcessType and TaskType at the metatype level which have process types and task types (clabjects at the type level) as their members. A process type is related to its task types by statements with property task with potencies 1-1. Property task is introduced at the metatype level with a mandatory and functional constraint for potencies 1-1 for its inverse property which ensures that every task type belongs exclusively to one process type.

Requirement (P2) is fulfilled through abstract metatype GatewayType, its properties in and out with dual potencies 2-2, and its concrete specializations SequenceType, AndSplitType, AndJoinType, OrSplitType, OrJoinType.

Task types are ordered through gateway types, i. e., members of clabject GatewayType at the type level, and in- and out-statements at the type level which are further instantiated at the occurrence level. For every gateway type, the in and out statements restrict the range of in and out for occurrences of that gateway type. This enforces at the occurrence level the order specified at the type level.

To ensure a correct process execution also considering the begin and end dates, one would have to set a deep constraint (not directly supported in DDM) in GatewayType that for each gateway occurrence (i. e. descendants of GatewayType at level *Occurrence*) the value of the property beginDate for each outgoing task occurrence (i. e. descendants of TaskType at level *Occurrence* connected to the gateway occurrence via out) is greater than or equal to the value of the property endDate of each incoming task occurrence.

Requirement (P3) is fulfilled through properties initialTask and finalTask which are specializations of property task and are used to model initial and final task types of process types as well as initial and final task occurrences of process occurrences. The initialTask- and finalTask-statements at the type level restrict the range for initialTask and finalTask at the occurrence level so that only occurrences of the initial and final task types may act as initial and final tasks. A mandatory and functional constraint ensures that every process type has exactly one initial task type. A mandatory constraint ensures that every process type has at least one final task type.

Requirement (P4) is fulfilled through property createdBy introduced between metatypes TaskType and ActorType, yet with source potency 1 and target potency 2, so that the ultimate instances of createdBy are cross-level links connecting task types with individual actors.

Requirement (P5) is fulfilled through property performedBy introduced at the metatype level with potencies 2-2 between clabjects TaskType and ActorType. At the type level, statements with property performedBy connect task types with actor types and thereby restrict the range of performedBy for the occurrence level. Occurrences of a task type may only be performed by an occurrence of an actor type when there is a corresponding statement at the type level.

A set of actor types may be authorized for one task type by a statement ensemble, that is, multiple performedBy statements with the same task type as source and different actor types as targets.

Requirement (P6) is fulfilled by a statement ensemble with property performedBy and potencies 1-0, connecting a task type to a set of individual actors, restricting for occurrences of that task type the range of performedBy. Occurrences of that task type may then only be performed by one or more of these individual actors. This is exemplified by task type Design, the occurrences of which may only be performed by BobBrown or LisaLoud (see Fig. 11).

Requirement (P7) is fulfilled through properties uses and produces introduced at the metatype level between clabjects TaskType and ArtifactType and potencies 2-2, meaning that these properties cover both the type level and the occurrence level. Statements at the type level restrict the properties' range at the occurrence level. For example,

task type Coding is associated with artifact type Code using property produces with potencies 1-1.

Requirement (P8) is fulfilled through property expectedDuration introduced between metatype TaskType and type Number with source potency 1 and target potency 1 (see Fig. 13). Together with a mandatory and functional constraint this specifies that every task type has exactly one numeric value for expectedDuration.

Requirement (P9) is fulfilled at the type level through abstract clabjects CriticalTask, SeniorActor, CriticalArtifact, ValidationTask, property validated introduced between ValidationTask and CriticalArtifact, the performedBy statement between CriticalTask and SeniorActor, and the produces statement between CriticalTask and CriticalArtifact. A functional and mandatory constraint on the inverse of validates for potencies 0-0 ensures that every critical artifact occurrence is validated by exactly one validation task occurrence. These statements and constraints express that every artifact produced by a critical task must be a critical artifact and as such must be validated by a validation task and that each critical task must be performed by a senior actor.

Requirement (P10) is fulfilled through clabject ProcessType at the metatype level (i. e., with potency 2) which represents both the class of process types and the class of process occurrences. Every process type, i. e., every instance of ProcessType, also acts as a class of process occurrences and can have arbitrary many process occurrences as members, in other words, the process type can be enacted multiple times.

Requirement (P11) is fulfilled through clabjects ProcessType and TaskType related through property task with potencies 2-2 at the metatype level which have process occurrences and task occurrences (clabjects at the occurrences level) as members of their members related by task-statements with potencies 0-0.

Requirement (P12) is fulfilled through properties beginDate and endDate introduced at the metatype level with source clabject TaskType and target clabject Date and with potencies 2-1 which means that the ultimate instances of these properties link task occurrences with particular dates. A mandatory and functional constraint for potencies 0-0 ensures that every task occurrence has exactly one start date and exactly one end date.

Requirement (P13) is fulfilled through properties uses, produces, and performedBy, which are introduced at the metatype level with potencies 2-2, meaning that their ultimate instances link task occurrences with artifact occurrences and individual actors, respectively.

Requirement (P14) is fulfilled by virtue of the semantics of statement ensembles in DDM. Multiple statements with property produces (or uses) with the same task type as source clabject and different artifact types as target clabjects restrict the range of produces (or uses) for occurrences of that task type to occurrences of those artifact types.

Requirement (P15) is not directly covered in the sense that DDM does not support multiple classification. In DDM, every concrete clabject at some lower level is instance of exactly one concrete clabject at the higher level.

In the absence of multiple classification, a typical workaround in two-level models it to use multiple inheritance at the type level by (i) defining a supertype for every actor type and (ii) defining a subtype for every permitted combination of permitted types. Using this approach in DDM, one would model actor types such as Designer and Tester first as abstract clabjects and then model all relevant combinations of actor types as concrete clabjects, e. g., concrete clabject DesignerAndTester specializing Designer and Tester, concrete clabject DesignerOnly specializing Designer. But such an approach is cumbersome as it entails a combinatorial explosion of types.

The given modeling situation is, however, in our opinion best supported by using *roles* as a modeling primitive. Roles (Gottlob et al. 1996) provide for dynamically adding and removing new instances of role types for an entity type, while at the same time supporting role-identity next to entity-identity, and allow for checking entity-equivalence of two roles of an entity. Roles can

be added as modeling primitive to a data model at the meta level if the data model supports deep properties, i. e., properties that are introduced at meta-types (or meta-classes) and are instantiated (by using the modeling primitive) at the type-level and are instantiated, recursively, at the instance level. This approach for *data model tailoring* using meta classes describing their classes and the instances of their classes has been shown for roles by (Klas and Schrefl 1995) and by (Dahchour et al. 2004). It can be used 1:1 in DDM to introduce *roles* at the meta type level and to model P15 at the type level.

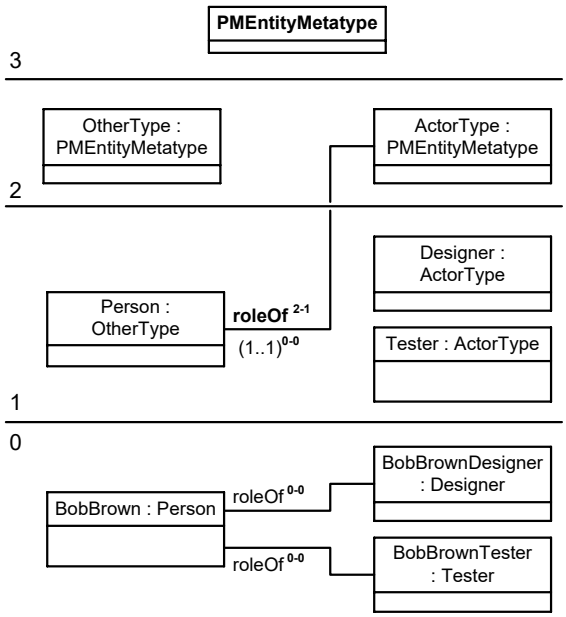

Figure 16: Modeling actors as roles played by persons

Alternatively, one can introduce roles for the single modeling situation of P15 as follows using DDM. Actors are modeled as reified roles played by a person (see Fig. 16). Person is modeled as first-order clabject. Metatype ActorType as source clabject introduces property roleOf with target clabject Person and source potency 2 and target potency 1. A mandatory and functional constraint for potencies 0-0 ensures that every actor occurrence is the role of exactly one person. The person BobBrown plays two roles, namely BobBrownDesigner, a member of actor type Designer, and BobBrownTester, a member of actor type Tester.

Requirement (P16) is not directly covered by DDM and would require a workaround in analogy to one of the workarounds described above.

Requirement (P17) is fulfilled through property performedBy introduced at the metatype level with potency 2-2, hence covering the type level as well as the occurrence level. Statements with property performedBy and potencies 1-1 relating task types with actor types restrict the range of performedBy, hence act as authorization for actors of these actor types.

Requirement (P18) is fulfilled through the possibility to generalize clabjects to abstract superclabjects. In contrast to the example given with (P18), DDM supports covariant specialization meaning that the range of performedBy becomes smaller in sub-clabjects and not broader.

Requirement (P19) is partially fulfilled through introducing with clabject PMEntityMetatype four 'last updated' properties (see Fig. 12) with source potencies ranging from 0 to 3, that is, one property for each instantiation level. Mandatory and functional constraints ensure that at every level every concrete clabject has a 'last updated' timestamp. Abstract clabjects, however, are not covered. An abstract clabject cannot have a property value that describes the abstract clabject itself.

Requirement (S1) is fulfilled at the type level through clabject ReqAnalysis associated to Analyst by a performedBy statement and by ReqAnalysis associated to ReqSpecification by a produces statement, both with potencies 1-1 (see Fig. 15). Cardinality constraints ensure that each occurrence produces only exactly one requirements specification and vice versa each specification is produced by exactly one occurrence of requirement analysis.

Requirement (S2) is fulfilled at the type level through clabject TestCaseDesign associated with SeniorAnalyst and TestCase by statements with properties performedBy and produces, respectively.

Requirement (S3) is fulfilled at the type level through task type Coding associated to artifact type Code by a produces statement and to actor type Developer by a performedBy statement. Furthermore, property employs is introduced with task type Coding as source clabject and ProgrammingLanguage as target clabject. Cardinality constraints ensure that every Code occurrence is linked to exactly one Coding occurrence, and every Coding occurrence is linked to one or more programming languages.

Requirement (S4) is fulfilled at the type level by property writtenIn introduced with artifact type Code as source clabject and ProgrammingLanguage as target clabject. A cardinality constraint ensures that every Code occurrence is written in at least one programming language.

Requirement (S5) is fulfilled at the occurrence level through abstract task occurrence CodingInCOBOL linked to programming language COBOL by a shared employs statement with potency 0-0 which is propagated to concrete CodingInCOBOL occurrences such as CodingFinanceApp.

Requirement (S6) is fulfilled at the occurrence level by abstract artifact occurrence COBOLCode linked to programming language COBOL by a shared writtenIn statement with potency 0-0 which is propagated to concrete COBOLCode occurrences such as FinanceAppCode.

Requirement (S7) is fulfilled at the occurrence level by abstract task occurrence CodingInCOBOL linked to developer AnnSmith by a shared performedBy statement which not only restricts the range of performedBy for concrete CodingInCOBOL occurrences such as CodingFinanceApp but is also propagated to them.

Requirement (S8) is fulfilled at the type level by task type Testing associated to artifact type TestReport by a produces statement.

Requirement (S9) is fulfilled by introducing property testedArtifact with source potency 1 and target potency 2 between artifact type TestReport as source clabject and metatype ArtifactType as target clabject. The dual potencies express that the ultimate instances of testedArtifact are at the occurrence level linking occurrences of TestReport with individual artifacts in their role as testedArtifact.

Requirement (S10) is fulfilled at the type level through abstract artifact type SoftwareEngineeringArtifact which introduces properties versionNo with potencies 1-1 and property responsibleActor with potencies 1-2. The latter associates clabject SoftwareEngineeringArtifact at the type level with clabject ActorType at the metatype level, yet the source potency 1 together with target potency 2 models that ultimate instances of responsibleActor link artifact occurrences to individual actors. Cardinality constraints ensure that every individual software engineering artifact has exactly one version number and exactly one responsible actor.

Requirement (S11-a) is not fulfilled by our solution. In DDM every concrete clabject at a lower level must be instance of exactly one concrete clabject at the next higher level. A workaround is sketched above in the discussion of fulfillment of requirement (P15). With this workaround, individual actor BobBrown would be modeled as an instance of clabject DesignerAndTester. Alternatively, as shown in Fig. 16, the different actor roles played by BobBrown are reified.

Requirement (S11-b) is fulfilled by linking type-level clabjects ReqAnalysis, TestCaseDesign, Coding and Testing with occurrence-level clabject BobBrown by createdBy statements.

Requirement (S12) is fulfilled at the type level by clabject Testing which has expectedDuration 9. Since expectedDuration is introduced by TaskType at the metatype level with potencies 1-1, its ultimate instances link task types with individual numbers.

Requirement (S13) is fulfilled at the type level by concrete task type TestCaseDesign which is a specialization of abstract task type CriticalTask. TestCaseDesign is associated by a performedBy-statement with potency 1-1 to actor type SeniorAnalyst, which is a specialization of abstract actor types Analyst and SeniorActor. Task type TestDesignReview, which is a specialization of ValidationTask, is associated with artifact type

TestCase, which is a specialization of CriticalArtifact, via a validates statement with potency 1-1. This, together with a cardinality constraint on the inverse of validates between CriticalArtifact and ValidationTask ensures that every test case occurrence is validated by exactly one occurrence of TestDesignReview.

6 Assessment of Modeling Solution

In this section we discuss the modeling solution from various directions with regard to different aspects of multi-level modeling. We also highlight different facets of DDM's modeling constructs, also with regard to related work.

6.1 Basic Modeling Constructs

DDM's basic modeling constructs are explained in Sect. 2. Let us now discuss some core characteristics of model elements in DDM.

The outstanding specific feature of DDM is that each domain concept, for example *Task* or *performed by*, is introduced only once in the multi-level model. The different incarnations of these concepts at different levels, for example *task occurrence* and *task type*, or, respectively, *'task occurrence performed by individual actor'* and *'task type performed by actor type'* and also *'task type performed by individual actors'*, are represented implicitly. These different incarnations of the concept at different levels—we also refer to them as facets of model elements—in turn have different facets, for example the instances of *'task type performed by actor type'* can be regarded as simple links but also as schema statements that impose constraints on statements on lower levels, such as on the allowed instances of *'task occurrence performed by individual actor'*. By virtue of dual potencies these originally implicit facets of model elements can be explicitly addressed, for example, performedBy with dual potencies 1-0 represents facet *'task type performed by individual actors'*. This characteristic of multi-level modeling elements is, of course, already partially realized in classical approaches to potency-based MLM (Atkinson and Kühne 2001) and more so with extensions to the basic approach (Lara et al. 2014), but it is only with dual potencies that this characteristic of multi-level model elements is fully realized.

This characteristic of DDM (and of other approaches to potency-based deep modeling) is also sometimes seen as a drawback hampering conceptual clarity. One may argue that what above we referred to as incarnations of domain concepts at different levels are actual domain concepts. For example, one may argue that the actual domain concepts are *task type* and *task* (the latter in the sense of task occurrence) which are level-specific and that there is no point in representing these two concepts as a single model element. We note, however, that in potency-based deep modeling approaches, explicitly defining both *task type* and *task* is possible if the modeler determines such constructs to be useful, but "one is not forced to do so" (Atkinson and Kühne 2008, p. 357). There are different MLM approaches in this direction based on power types (Carvalho and Almeida 2018; Jeusfeld and Neumayr 2016) where these level-specific domain concepts (or different incarnations of a domain concept at different levels) are explicitly represented as model elements at different levels connected by dependencies such as *partitions* or *is most general instance of*.

6.2 Levels

In DDM, in contrast to other potency-based MLM approaches, the *level*, the *order*, and the *potency* of a clabject always match. For example, a 2nd-order clabject must be at level 2 and must have potency 2. In essence, one of the three terms would suffice for DDM, but for historical reasons, we stick with this terminology and use the three terms basically synonymously.

Clabjects in a clabject hierarchy are organized into levels simply based on the number of instantiation steps they are away from the root clabject of that hierarchy. In addition to their potency number, levels have level names which can be used together with clabject names to refer to the different kinds of clabjects at the different levels. 'Instance-of' is the only level-determining relationship and every

Table 1: Fulfillment of rules and requirements by the presented modeling solution. Dual deep modeling supports most of the required features out of the box (✓) while others require language extensions and workarounds (~) .

Req.	Status	Remark
P1	✓	
P2	✓	
P3	✓	
P4	✓	
P5	✓	
P6	✓	
P7	✓	
P8	✓	
P9	✓	
P10	✓	
P11	✓	
P12	✓	
P13	✓	
P14	✓	
P15	~	
P16	~	
P17	✓	
P18	✓	covariant specialization
P19	✓	without abstract clabjects
S1	✓	
S2	✓	
S3	✓	
S4	✓	
S5	✓	
S6	✓	
S7	✓	
S8	✓	
S9	✓	
S10	✓	
S11a	~	
S11b	✓	
S12	✓	
S13	✓	

clabject except for the clabjects in the highest level have exactly one 'instance-of' relationship to a clabject in the next higher level.

In some MLM approaches a level of a multi-level model is a self-contained model or a kind of module within the multi-level model. This is not the case for DDM. In DDM, a level simply collects clabjects of the same order.

A characteristic of DDM is perhaps the flexibility and openness with which we deal with such central concepts such as 'instantiation' and 'instantiation level' and concepts derived from them such as 'potency' and 'order'. In our solution to the process challenge we used conventional instantiation levels labeled *occurrence*, *type*, and *metatype* but DDM is open to other flavors of instantiation, sometimes also referred to as concretization (Frank and Töpel 2020; Neumayr et al. 2009), giving rise to, for example, levels in a product hierarchy labeled *category*, *model*, and *physical entity*. Yet, independent of the flavors of instantiation, DDM clabject hierarchies are always to be interpreted in a multi-faceted manner. For example, every clabject at level 2, e. g., product category *Car*, has an object facet, e. g., describing the product category itself, a class facet, e. g., representing the class of car models, and a metaclass facet, e. g., representing the class of individual physical cars partitioned by car model.

6.3 Number of Levels

Our solution to the modeling challenge comes with four levels with the top-most level, the metametatype level with potency 3, only containing PMEntityMetatype as root clabject of the hierarchy.

An alternative solution would be a hierarchy with only three levels with PMEntityType as abstract 2nd-order clabject generalizing clabjects TaskType, ProcessType, GatewayType, ActorType and OtherType. Apart from adapting PMEntityMetatype to PMEntityType and replacing instantiation relationships to PMEntityMetatype with specialization relationships to PMEntityType, this would not entail any further changes to the solution.

Another alternative solution would comprise separate clabject hierarchies for tasks, actors, gateways, artifacts, each with three levels and with TaskType, ActorType, GatewayTyp, and ArtifactType as root clabject of their respective hierarchy. When modeling in this flavor, programming languages would be modeled in a separate two-level clabject hierarchy with first-order clabject ProgrammingLanguage as root clabject. Such a solution would make obsolete the third-order clabject PMEntityMetatype and the second-order clabject OtherType. Apart from that, this alternative solution would not entail any further changes, the properties and statements would be exactly the same.

Another intuitive solution in DDM would be to model processes, tasks, and gateways in one clabject hierarchy with PMEntityMetatype as root clabject, and to model the other parts in separate hierarchies, one with root clabject ArtifactType, another one with root clabject ActorType, and yet another one with root clabject ProgrammingLanguage. For these other clabject hierarchies we would give a different name to the 0-level, namely 'Individual' instead of the more process-specific 'Occurrence'.

The choice for one of the three variants has hardly any serious advantages and disadvantages. We assume, however, that experienced DDM modelers would rather choose the latter variant with multiple clabject hierarchies, since then classes and metaclasses like OtherType are no longer necessary and levels can be named more accurately per hierarchy.

Since we treat atomic data types and values also as clabjects, our solution assumes another clabject hierarchy with clabject SimpleData as root clabject and instantiation levels named Type and Value.

6.4 Cross-Level Relationships

In DDM, relationships (associations and links) can cross level boundaries without generic restrictions. A relationship can connect source and target clabjects of different orders (i. e., clabjects at different levels) and can come with different source and target potencies. The only restriction is that the source potency is not higher than the order of the source clabject and that the target potency is not higher than the order of the target clabject.

When introducing a property, the source clabject and source potency together with the target clabject and target potency specify which clabjects can be linked by ultimate instances of the property. For example, the ultimate instances of property createdBy, which is introduced between source clabject TaskType with source potency 1 and target clabject ActorType with target potency 2. TaskType is a second-order clabject respresenting task types as well as task occurrence, but because of source potency 1, property createdBy can only be instantiated down to task types and an ultimate instance of createdBy links a task type to an individual actor.

A somehow different kind of cross-level relationships is exemplified by properties testedArtifact and responsibleActor which are introduced between source and target clabjects at different levels but by virtue of source and target potencies their ultimate instances are at the same level. For example, property testedArtifact is introduced between first-order source clabject TestReport and second-order target clabject ArtifactType with source potency 1 and target potency 2 so that ultimate instances of property testedArtifact link occurrences of TestReport with individual artifacts, the latter without restriction to a specific artifact type.

Another somehow different kind of cross-level relationships are intermediate cross-level relationships exemplified by the performedBy statement ensemble with source potency 1 and target potency 0 of first-order source clabject Design (see Fig. 11) which restricts the range of performedBy of occurrences of Design to the set of BobBrown and LisaLoud.

6.5 Cross-Level Constraints

Properties and statements with potencies above 1 naturally span multiple levels, that is why we also call them deep properties and deep statements in DDM. Every statement with a property connecting

clabjects at a higher levels constrains the range of this property for descendant clabjects at all lower levels spanned by the property. Cardinality constraints are specified together with dual potencies to indicate the facet of the property which should be constrained.

6.6 Integrity Mechanisms

The DDM approach comes with an implementation in F-Logic (see Neumayr et al. 2018), the main application of which is to check explicit (such as cardinality constraints) and implicit (such as range restrictions) integrity constraints.

6.7 Deep Characterization

In DDM, the source and target potencies of a property or statement indicate the depth of characterization. A property or statement with source potency 2 and target potency 3 spans the next two lower levels below the source clabject and the next three lower levels below the target clabject. Design choices made at higher levels cannot be overridden at lower intermediate levels.

6.8 Generality

DDM facilitates the modeling of highly domain-specific multi-level hierarchies with different number of levels and different level names for different hierarchies (as exemplified in Neumayr et al. 2018). Yet for solving the process challenge we opted for a more general level structure with a single clabject hierarchy with a uniform set of levels named Metametatype, Metatype, Type, and Occurrence. This level structure is mostly domain-independent.

With regard to the generality of the solution for the process modeling domain, one could say that the metatype part of the solution –especially clabjects ProcessType, TaskType, and GatewayType with their deep properties– embodies invariant principles of the process modeling domain with minimal redundancy covering both the modeling of process types as well as process occurrences.

6.9 Extensibility

It is straightforward to extend the solution with further task types and actor types and to relate these. They can be introduced without abstract types as generalizations or as specializations of abstract task types such as CriticalTask or ValidationTask or as specialization of abstract actor types SeniorActor or Analyst.

In DDM, when the need of specializing a concrete clabject C arises, one always has the choice between introducing abstract clabjects (which are modeled as abstract instances of C) at the next lower level or by refactoring C into an abstract clabject and introduce the specializations of C as concrete sub-clabjects.

6.10 Formalization and Tool Support

The structure and semantics of DDM is formally defined in F-Logic by a vocabulary, deductive rules, and constraints (see Neumayr et al. 2018). Together with an F-Logic engine like Flora-2/ErgoAI this formalization can be used for checking the integrity of and for querying a multi-level model.

Future work will focus on formulating the challenge solution in terms of DDM's F-Logic formalization. We will also include counter examples violating integrity constraints. We will then use the F-Logic engine to query and to check the integrity of the multi-level process model hierarchy. That formalization of our modeling solution in F-Logic together with performance reports and validation results will be made available open source.

7 Related Work

We first discuss the presented solution in the context of multi-level modeling in general. We then compare our DDM solution to the multi-level process challenge with solutions to the MULTI 2019 process challenge (Almeida et al. 2019). An overview of the comparison is given in Tab. 2.

7.1 Multi-Level Modeling

The DDM approach relies on the Orthogonal Classification Architecture (OCA) and the notion of clabject in connection with potency-based deep instantiation. The OCA distinguishes between a linguistic classification dimension and an ontological classification dimension (Atkinson and Kühne 2003). The elements of the multi-level process model are linguistic instances of Clabject. The clabject ProcessType, for example, is an ontological instance of PMEntityMetatype. In this regard, the notion of clabject is central (Atkinson 1997): A clabject has a class facet and an object facet. Unlike traditional deep instantiation (Atkinson and Kühne 2001), DDM distinguishes between a source potency and a target potency.

An important further difference of DDM to the traditional OCA is that DDM is flexible with regard to the level-segregation principle, see the discussion in Sect. 6.2. One can use DDM not only for modeling multi-level classification hierarchies but also for modeling concretization hierarchies as featured by the m-object approach (Neumayr et al. 2009) and by FMML^x (Frank 2014). DDM's multifaceted interpretation of clabjects, see Sect. 4.1.1, is equally suited to classification and concretization hierarchies. In addition, DDM offers a specific naming scheme that is especially suitable for concretization hierarchies. In this paper, however, for solving the process challenge, we opted for using classification as level-segregation principle and did not make use of DDM's specific naming scheme.

A multi-level object (m-object) describes a hierarchy of objects at multiple levels of abstraction (Neumayr et al. 2009). A concretization mechanism allows for the refinement of the data model. The concept of multi-level business artifact (MBA) extends the notion of m-object for artifact-centric business process modeling (Schuetz 2015). The artifact-centric (or data-centric) approach to business process modeling focuses on the data objects and the life cycles of those objects. An MBA is an m-object that comprises a life cycle model for each level of abstraction. More specific levels may specialize life-cycle models via the mechanism of behavior-consistent specialization (Schrefl and Stumptner 2002), leading to the notion of hetero-homogeneous business process models.

7.2 Contributions to the Multi-Level Process Challenge

Rodríguez and Macías (2019) employ MultEcore, an extension of the Eclipse Modeling Framework for multi-level modeling (Macías et al. 2016), to solve the multi-level process challenge. The MultEcore solution distinguishes between an *application hierarchy* and two *supplementary hierarchies*. The application hierarchy has four levels: general process model elements, domain-specific process, enterprise-specific process, and process instances, which is similar to our solution in DDM. The supplementary hierarchies consist of *language supplementary hierarchy* and technology supplementary hierarchy, which are orthogonal to the application hierarchy and serve for aspect-oriented modeling.

Like DDM, MultEcore is a potency-based multi-level modeling approach. There are, however, several key differences between MultEcore and DDM regarding the use of potencies. MultEcore models specify a *minimum potency* and a *maximum potency* for elements, relationships, and attributes. In addition, for elements and relationships, MultEcore models specify a *depth*. The minimum and maximum potencies define the first and last level, respectively, where an element, relationship, or attribute can be instantiated. The depth defines at how many levels below the element or relationship can be instantiated.

The MultEcore solution employs Multilevel Coupled Model Transformations (MCMTs) for the specification of cross-level constraints. In particular, an MCMT realizes the requirement that an actor must be authorized to execute a certain task (P17). The DDM solution realizes that constraint using deep statements.

Somogyi et al. (2019) employ the dynamic multi-layer algebra (DMLA)[1] to provide a solution to the multi-level process challenge. Unlike

[1] http://www.aut.bme.hu/Pages/Research/VMTS/DMLA

Table 2: Comparison of the characteristics of the different multi-level modeling approaches used in solutions for the multi-level process challenge

Solution	Levels	Potencies	Constr. Lang.	Tool
MultEcore	Explicit	Min/Max	MCMT	EMF Extension
DMLA	None	None	Imperative	MLM Playground
DeepTelos	Implicit	None (Powertype)	Datalog-based	ConceptBase
DDM	Explicit	Source/Target	None (F-Logic)	Flora-2/ErgoAI

the MultEcore and the DDM solutions, DMLA does not have explicit levels. DMLA has a flexible instantiation mechanism, which resembles *refinement* from traditional two-level modeling.

The DMLA solution to the multi-level process challenge distinguishes between *structural instantiation* and *specifying-instantiation*, although specifying-instantiation is not directly supported by the DMLA solution.

The DMLA solution employs an imperative constraint language for enforcement of certain rules. For example, the requirement that COBOL code is written in the COBOL language is realized using a simple imperative constraint.

Jeusfeld (2019) proposes a solution to the multi-level process challenge using DeepTelos (Jeusfeld and Neumayr 2016), an extension of Telos for multi-level modeling based on powertypes (Odell 1998). Telos (Koubarakis et al. 2021) is a knowledge representation language that serves as the fundamental for the ConceptBase deductive database system (Jarke et al. 1995). In Telos, every object can instantiate any other object and, in turn, be instantiated by any other object. Dual deep instantiation, the precursor of DDM, which we employ in this paper, also has an implementation in ConceptBase.

The DeepTelos solution to the multi-level process challenge does not explicitly model levels. While not explicitly modeled, however, levels can still be identified. The Omega level comprises the Proposition object, which is the most fundamental concept in Telos; the core semantics of DeepTelos are defined over Proposition. The M3 level comprises the basic elements for the representation of graph-based models, i. e., NodeOrLink and its specialization Node, which the elements of process models are specializations of. The M2 level comprises general concepts for process modeling; the DeepTelos solution reuses a definition in Telos of the BPMN language (see OMG 2013 for more information on BPMN). The M1 level comprises the specification of the Acme software development process. The M0 level comprises instantiations of specific process models.

Instead of potencies, which are employed in DDM, DeepTelos has the concept of *most-general instances*, which realizes the 'powertype' concept. For example, the DeepTelos solution comprises ProcessType with most-general instance Process, ProcessElementType with ProcessElement, TaskType with Task, ActorType with Actor, and ArtifactType with Artifact.

The DeepTelos solution also employs declarative constraints based on Datalog to realize cross-level constraints for specification of authorization of actors for certain tasks. In the DDM solution, those constraints are implicitly covered by the instantiation and specialization mechanisms, although using the F-Logic formalization, DDM models could be complemented by F-Logic rules.

8 Conclusions

We have presented a solution to the multi-level process challenge using DDM. The solution demonstrates the strength of DDM, namely the compact modeling of domains that span multiple instantiation levels and require level-crossing relationships. We conclude the paper with lessons learned and implications for future work.

- When looking for a solution for requirement (P19) we missed a language feature to model properties that should be instantiated independently of instantiation levels and independently of the distinction between abstract and concrete clabjects.

  In future work, we will investigate how to extend DDM with such *level-independent properties*. We are thinking of either a mix-in mechanism for metadata or of extending DDM's linguistic metamodel accordingly.

- Looking at the type level of the solution (see Fig. 13), we see a lot of mandatory and functional constraints, especially at both ends of all `in` and `out` statements. What we missed when developing the solution was a language feature to specify at the metatype level, that every concrete `in` statement and every concrete `out` statement at the type level comes with such constraints.

  In future work, we will investigate how to extend DDM with generic constraints. What we are thinking of is a flexible mechanism that lets the modeler specify with a property a generic constraint (such as a mandatory constraint for potencies 0-0) together with dual potencies indicating the intermediate levels at which statements of that property should be supplemented with constraints based on that generic constraint. In this respect, the mechanisms for *deep multiplicities* (Atkinson et al. 2015) are very promising and we will investigate how they can be combined with dual potencies.

- Requirements (P15) and (P16) highlighted DDM's lack of a mechanism for multiple classification. The restriction that a clabject has at most one clabject as its class is an integral part of DDM's linguistic metamodel and contributes to DDM's overall simplicity.

  We will investigate in future work the impact of lifting this restriction and alternatively provide solutions based on data model tailoring as sketched in Sect. 5.

References

Almeida J. P. A., Rutle A., Wimmer M., Kühne T. (2019) The MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, pp. 164–167

Atkinson C. (1997) Meta-modelling for distributed object environments. In: Proceedings First International Enterprise Distributed Object Computing Workshop, pp. 90–101

Atkinson C., Gerbig R., Kühne T. (2015) A unifying approach to connections for multi-level modeling. In: Lethbridge T., Cabot J., Egyed A. (eds.) 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015. IEEE Computer Society, pp. 216–225

Atkinson C., Kühne T. (2001) The Essence of Multilevel Metamodeling. In: Gogolla M., Kobryn C. (eds.) «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. Lecture Notes in Computer Science Vol. 2185. Springer, pp. 19–33

Atkinson C., Kühne T. (2003) Model-driven development: A metamodeling foundation. In: IEEE software 20(5), pp. 36–41

Atkinson C., Kühne T. (2008) Reducing accidental complexity in domain models. In: Software & Systems Modeling 7(3), pp. 345–359

de Carvalho V. A., Almeida J. P. A. (2018) Toward a well-founded theory for multi-level conceptual modeling. In: Softw. Syst. Model. 17(1), pp. 205–231

Dahchour M., Pirotte A., Zimányi E. (2004) A role model and its metaclass implementation. In: Inf. Syst. 29(3), pp. 235–270

Frank U. (2014) Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. In: Bus. Inf. Syst. Eng. 6(6), pp. 319–337

Frank U., Töpel D. (2020) Contingent level classes: Motivation, conceptualization, modeling guidelines, and implications for model management. In: Guerra E., Iovino L. (eds.) MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020, Companion Proceedings. ACM, 86:1–86:10

Gottlob G., Schrefl M., Röck B. (1996) Extending Object-Oriented Systems with Roles. In: ACM Trans. Inf. Syst. 14(3), pp. 268–296

Hürsch W. L. (1994) Should Superclasses be Abstract? In: Tokoro M., Pareschi R. (eds.) Object-Oriented Programming, Proceedings of the 8th European Conference, ECOOP '94, Bologna, Italy, July 4-8, 1994. Lecture Notes in Computer Science Vol. 821. Springer, pp. 12–31

Jarke M., Gallersdörfer R., Jeusfeld M. A., Staudt M. (1995) ConceptBase - A Deductive Object Base for Meta Data Management. In: Journal of Intelligent Information Systems 4(2), pp. 167–192

Jeusfeld M. A. (2019) DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, pp. 66–77

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level Modeling with Most General Instances. In: Comyn-Wattiau I., Tanaka K., Song I., Yamamoto S., Saeki M. (eds.) ER 2016. LNCS Vol. 9974, pp. 198–211

Klas W., Schrefl M. (1995) Metaclasses and Their Applications, Data Model Tailoring and Database Integration. Lecture Notes in Computer Science Vol. 943. Springer

Koubarakis M., Borgida A., Constantopoulos P., Doerr M., Jarke M., Jeusfeld M. A., Mylopoulos J., Plexousakis D. (2021) A retrospective on Telos as a metamodeling language for requirements engineering. In: Requirements Engineering 26(1), pp. 1–23

de Lara J., Guerra E., Cobos R., Moreno-Llorena J. (2014) Extending Deep Meta-Modelling for Practical Model-Driven Engineering. In: Comput. J. 57(1), pp. 36–58

Macías F., Rutle A., Stolz V. (2016) MultEcore: Combining the Best of Fixed-Level and Multi-level Metamodelling. In: Atkinson C., Grossmann G., Clark T. (eds.) Proceedings of the 3rd International Workshop on Multi-Level Modelling. CEUR Workshop Proceedings Vol. 1722. CEUR-WS.org, pp. 66–75 http://ceur-ws.org/Vol-1722/p6.pdf

Neumayr B., Grün K., Schrefl M. (2009) Multi-Level Domain Modeling with M-Objects and M-Relationships. In: Kirchberg M., Link S. (eds.) Proc. Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM 2009). CRPIT Vol. 96. Australian Computer Society, pp. 107–116 http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV96Neumayr.html

Neumayr B., Schuetz C. G., Jeusfeld M. A., Schrefl M. (2018) Dual deep modeling: Multi-level modeling with dual potencies and its formalization in F-Logic. In: Software and Systems Modeling 17(1), pp. 233–268

Odell J. J. (1998) Power Types In: Advanced Object-Oriented Analysis and Design Using UML Cambridge University Press

OMG (2013) Business Process Model and Notation (BPMN) – Version 2.0.2 https://www.omg.org/spec/BPMN/

Rodríguez A., Macías F. (2019) Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.)

22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, pp. 152–163

Schrefl M., Stumptner M. (2002) Behavior-Consistent Specialization of Object Life Cycles. In: ACM Transactions on Software Engineering and Methodology 11(1), pp. 92–148

Schuetz C. G. (2015) Multilevel Business Processes – Modeling and Data Analysis. Springer

Somogyi F. A., Mezei G., Urbán D., Theisz Z., Bácsi S., Palatinszky D. (2019) Multi-level Modeling with DMLA - A Contribution to the MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, pp. 119–127