Evaluating DeepTelos for ConceptBase

A Contribution to the Multi-Level Process Challenge

Manfred A. Jeusfeld *,a

[a] University of Skövde, School of Informatics, Box 408, Sweden

Abstract. *The process modeling challenge provides an opportunity to compare various approaches to multi-level conceptual modeling. In particular, the challenge requests the definition of constructs for designing process models plus the facilities to create process models with these constructs, and to analyze the execution of such processes, all in one multi-level model. In this paper, we evaluate the performance of DeepTelos in solving the challenge. DeepTelos is an extension of the Telos modeling language that adds a small number of rules and constraints to the Telos axioms in order to facilitate multi-level modeling by means of so-called most-general instances, a variant of the powertype pattern. We present the technology behind DeepTelos and address the individual tasks of the process modeling challenge. A critical review discusses strengths and weaknesses exposed by the solution to the challenge.*

Keywords. Multi-level modeling • Telos • Process modeling • ConceptBase • Datalog

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

1 Introduction

Telos (Mylopoulos et al. 1990) is a knowledge representation language used for conceptual metamodeling in particular within the domain of requirements engineering (Koubarakis et al. 2021). In its original definition, it features unlimited instantiation hierarchies (tokens/objects, classes, metaclasses, metametaclasses, etc.) alongside a so-called omega-level of classes, whose instances can stem from any of the aforementioned levels. While the metamodeling capabilities of Telos are quite flexible, in particular by allowing relations spanning different instantiation levels, it does suffer from the accidental complexity of classical object-oriented modeling languages (Atkinson and Kühne 2008). For this reason, we investigated a number of options to add multi-level modeling features to Telos without compromising the other features of Telos.

The purpose of this paper is to evaluate the expressive power of DeepTelos as a multi-level modeling language. The evaluation is performed via the creation of a solution to a generic multi-level modeling challenge, which presents a number of requirements for a multi-level process modeling framework. Some of these requirements are about the formalization of process modeling constructs, while others are about the use of the constructs to define process models and their executions.

The paper is organized as follows. First, we present the features of Telos as implemented by the ConceptBase system (Jeusfeld 2009, 2022). We then review an early attempt to represent process models with Telos (Jarke et al. 1990) and highlight its weaknesses with respect to the accidental complexity and its problems to support process execution analysis. We then discuss the definition of DeepTelos via a small number of deductive rules and integrity constraints.

* Corresponding author.
E-mail. manfred.jeusfeld@acm.org
Note: This work is an extension of the article "DeepTelos for ConceptBase: A contribution to the MULTI process challenge", which was published in the proceedings of MODELS/MULTI 2019.

The main body of the paper presents the detailed solution to the multi-level process modeling challenge of this special issue. We conclude by a discussion of the strengths and weaknesses of DeepTelos for solving the modeling challenge.

2 Constructs of Telos and DeepTelos

The basic constructs of Telos[1] are instantiation, specialization, and attribution/relations. They are defined by a set of axioms (Jeusfeld 2009) with the following semantics:

Instantiation: An object x (called the instance) can be assigned to a class c by a fact (x in c). Due to the Datalog semantics (Ceri et al. 1989) of Telos, the set of instances of a class (its extension) is always finite. An instance of a class can use the attributes defined at any of its classes by instantiating one or all of the attributes of the class. An object can have any number of classes. The predicate (x in c) can be used in the condition and the conclusion position of a deductive rule. Hence, instantiation can be derived by deductive rules.

Specialization: An object c can be defined as a specialization of an object d, denoted by (c isA d). Any instance of c is also an instance of d by a predefined axiom. We call the object c the subclass of d, and d the superclass of c. Subclasses can refine the attributes of superclasses by specializing the attribute class of the corresponding attribute at the superclass. The predicate (c isA d) may not be derived by user-defined rules, i.e. its semantics is completely defined by the fixed axioms of Telos.

Attribution/relations: Two objects x and y can be linked by an attribute. Telos regards values (numbers, strings, etc.) as objects. Hence, it does not distinguish classical attributes from (binary) relations. The predicate (x m y) expresses that there is an attribution link between x and y, where m is the label of some attribute definition of some class of x. Explicit attributes or relations are also objects in Telos.

There are five predefined objects in Telos. The object "Individual" has all objects as instances that are not explicit attributes, instantiations, or specializations, i.e. all node-like objects. The object "Attribute" has all explicit attributes/relations as instances, the class "InstanceOf" has all explicit instantiations as instances, and "IsA" all explicit specializations. The predefined object "Proposition" subsumes the instances of all four previous classes. The five predefined objects form the so-called "omega" level of Telos. These classes have instances that can be on any ontological abstraction level. The full specification of Telos as implemented by ConceptBase is provided in (Jeusfeld M. A. et al. 2021, pp. 14ff).

DeepTelos adds three constructs to Telos, the "most-general instance" relation, the derivable specialization construct, and the enumeration construct:

Most-general instance: This construct declares the object m as "most-general instance" of the class c, expressed by a fact (m IN c). Any instance of c shall then be a subclass of m, and subclass of m shall be an instance of c.[2] As consequence of the DeepTelos rules (m IN c) implies (m in c) because the isA-relation is reflexive in Telos.

Derivable specialization: This is a second construct to declare c as derived specialization of d, expressed by (c ISA d).

Enumerations[3] : This construct is the counterpart to UML enumerations. It provides an easy way to declare a class by means of a fixed set of instances. In DeepTelos, we use enumerations to lift a finite set of instances one level up, much like with singleton classes.

[1] In the following we use the term Telos as referring to the version of Telos as implemented by ConceptBase. Koubarakis et al. (2021) discusses some of the differences of this version of Telos compared to its original specification.

[2] This construct is the inverse of the powertype construct used in MLT*. Hence, (m IN c) is equivalent to (c powertypeOf m).

[3] Enumerations were not part of the first specification of DeepTelos. The construct is introduced here to further reduce the accidental complexity of DeepTelos models.

The three DeepTelos constructs and the five associated multi-level rules and one constraint listed in Sect. 2.2 realize a simple multi-level modeling environment. Basic constructs are defined as attributes of the omega class "Proposition" in Telos/ConceptBase. Since this class can be extended by a user of ConceptBase, any user can add new modeling constructs. In the case of this challenge, we defined an attribute "lastupdate" for "Proposition". This attribute of "Proposition" facilitates the representation of the last update time for any explicit fact in a model. Likewise, the DeepTelos constructs for most-general instances are defined at the omega class "Proposition". Consequently, traditional classes as well as attributes and relations can be most-general instances.

The most-general instance construct is a flavor of the powertype pattern. Partridge et al. (2018) survey multiple powertype flavors used in information systems engineering. The closest flavor in that survey to our approach is the "UML Powertype". The difference lies in the semantic grounding. While Partridge et al. (2018) use set-theoretic specifications, DeepTelos uses the minimal model semantics of Datalog. Specifically, DeepTelos needs two deductive rules to (1) derive specializations from instantiations to the powertype, and (2) derive instantiations to the powertype from specializations of the (most-general) instance of the powertype. DeepTelos does not offer different types of specializations such as complete and disjoint decomposition either.

2.1 The 1990 Software Process Model

In the late 1980-ties, Telos was used as the foundation of a repository system ConceptBase (Jarke et al. 1995) to store artifacts created during software development. The data model for the repository is described by Jarke et al. (1990). It purely uses the metamodeling features of Telos.

The process model supported three abstraction levels. The top level defines the metaclasses "DesignObjectType", "DesignDecisionType", and "DesignToolType". The relations "from/to" are for establishing the provenance of design objects. The middle level defines the set of design decision types supported by the repository. Here, the design decision type "EntHierMapMoveDown" denotes a strategy to map specialization hierarchies to relational tables. The input design objects ("tdlentities") are class definitions in TDL (Taxis Design Language). The result ("nonfirstrelations") of the mapping are relational table definitions (design object type "DBLP_REL_DO"). The design tool type is "MappingAssistant". The lowest level shows an example mapping of two classes ("Papers", "Invitations") to a relational table definition.

The example shows that Telos treats explicit attributes/relations like ordinary objects. For example, the "nonfirstrelations" relation of "EntHierMapMoveDown" is an instance of the "to" relation of "DesignDecisionType". In the textual syntax of Telos, the model is represented as:

```
DesignDecisionType with attribute
    from: DesignObjectType;
    to: DesignObjectType; by: DesignToolType
end

EntHierMapMoveDown in DesignDecisionType with
  from tdlentities: TDL_EC_DO
  to nonfirstrelations: DBLP_REL_DO
  by tool: MappingAsistant
end
```

The software process model can represent the provenance of design objects. However, the categories "from", "to", and "by" are not available to formulate queries at the lowest abstraction level, e.g. to show the provenance of design objects like "InvitationsRel_0". This is due to the way relations such as "to" are instantiated in Telos. At the next lower abstraction level, a new relation like "nonfirstrelations" needs to be defined. While it could use the same name "to", this is not a general solution, since instances of "DesignDecisionType" can have several relations with the category "to". As a consequence, the original software process model realized with Telos led to redundant and unnecessarily complex query definitions to analyze the provenance of design objects.

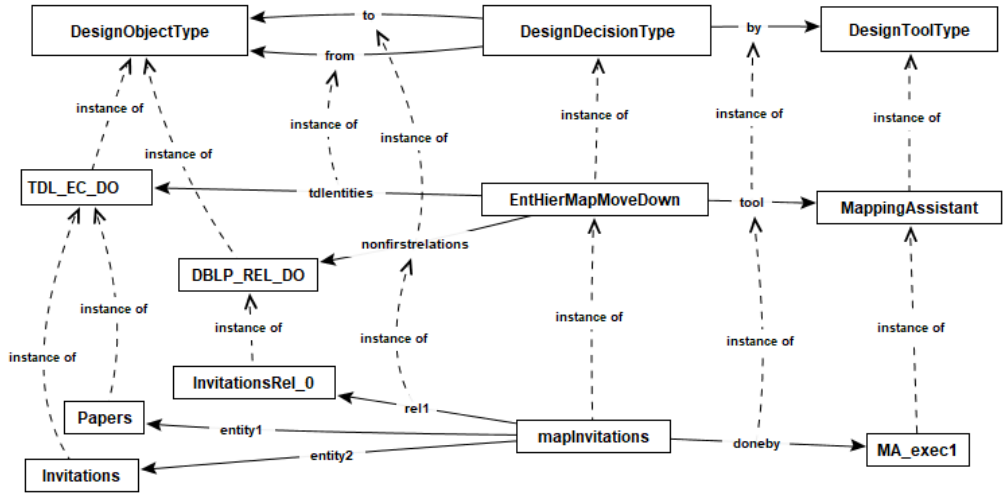

Figure 1: Software process model adapted from Jarke et al. (1990, p. 98).

2.2 DeepTelos: An extension for Telos to support multilevel modeling

DeepTelos (Jeusfeld and Neumayr 2016) was originally defined by just three deductive rules extending the existing Telos constructs attribution, instantiation, and specialization. These three rules were later extended to five rules and one constraint to better integrate the derived specializations of DeepTelos and the existing Telos specialization axioms. The core idea of DeepTelos is to employ the powertype pattern (Gonzalez-Perez and Henderson-Sellers 2006; Odell 1998) via the following rule: if a class *c* has a "most-general instance" *m*, then any instance of *c* is a subclass of *m* (see "mrule4" in Sect. 2.2). The most-general instance *m* serves as a "proxy" for class *c* at one instantiation level lower. It has all instances of instances of *c* as its instances. The class *m* itself can have another most-general instance *m*1, which serves as a proxy for *m* at even one instantiation level lower. So, the lattice of most-general instance relations, denoted as (m IN c), spans a family of modeling levels. ConceptBase (Jarke et al. 1995) is a multi-user database system for managing all kinds of models and metamodels. It implements its logical component (rules, constraints, queries) via a Datalog-neg engine. It also features a graphical user interface. Formally, most-general instances are defined by the extension of the predicate (m IN c) under Datalog-neg semantics of the rules defined below in this subsection.

DeepTelos Revision 2 is defined by five deductive rules and one constraint. The rules are defined in terms of two new constructs IN and ISA defined for any proposition (see source code in Sect. 10.1). The first rule is the main rule: If there is a most-general instance *m* of *c* (m IN c) and instance *x* of *c* and *x* is not already derivable to be a (Telos) specialization of *m*, then (x ISA m) is derived, i.e. all such instances become (DeepTelos) specializations of the most-general instance *m*. The arrow symbol in the formula stands for the logical implication, separating the rule body from the real head.

```
mrule1: forall m,x,c/Proposition
  (x in c) and (m IN c) and not (x isA m)
    ==> (x ISA m)
```

The second rule realizes the class membership inheritance from sub classes to super classes. A virtually identical rule is predefined for the Telos specialization predicate (c isA d). We use (c ISA d) in DeepTelos, because the Telos specialization predicate may not be derived itself.

```
mrule2: forall x,c,d/Proposition
  (c ISA d) and (x in c)  ==> (x in d)
```

The third rule links two most-general instances by a specialization relation if their classes are also specialized. This rule allows managing parallel hierarchies of most-general instances.

```
mrule3: forall c,d,m,n/Proposition
   (m IN c) and (n IN d) and (c ISA d)
     ==> (m ISA n)
```

The next rule is the reverse case of "mrule1": if a most-general instance *m* has a subclass *x*, then this is also an instance of the powertype *c* of *m*. In the ConceptBase implementation, we demand for technical reasons that *x* is not a so-called query class.

```
mrule4: forall m,x,c/Proposition
   (m IN c) and (x isA m)  ==> (x in c)
```

The fifth rule is a variant of "mrule4" that takes into account that we have two different predicates for specialization in DeepTelos.

```
mrule5: forall m,mx,x,c/Proposition
   (m IN c) and :(x isA mx): and (mx ISA m)
      ==> (x in c)
```

The predicate :(x isA mx): stands for an explicitly declared specialization. Finally, DeepTelos has one constraint to prevent inconsistent hierarchies of most-general instances:

```
mconstr1: forall x,m,c/Proposition
  (m IN c) and (x in c) ==> not (x in m)
```

DeepTelos is similar to MLT* (Fonseca et al. 2018) wrt. the use of the powertype construct. The difference to MLT* is that DeepTelos has far fewer axioms. The additional axioms of MLT* do make sense to prevent modeling errors but we decided not to realize them in DeepTelos for the sake of simplicity. MLT-Telos (Jeusfeld et al. 2020) is however available as a comprehensive implementation of MLT* in ConceptBase.

A notable difference between DeepTelos and MLT* is the reification of relations and attributes. In DeepTelos, attributes/relations can be defined as most-general instances of attributes/relations at the next higher instantiation level. DeepTelos only supports a single subclass hierarchy under a given most-general object. So, dual decompositions like "persontype by gender" or "persontype by age group" are not supported. This is due to the simplicity of the DeepTelos rules, in particular "mrule2" and "mrule4" (see Sect. 2.2). We have derived the variant MLT-Telos that incorporates an implementation of MLT*, which itself can cope with such multiple decompositions via the "partitions" construct. This paper is however using DeepTelos and thus does not support multiple decompositions of a most-general instance.

2.2.1 Enumeration classes

The multi-level modeling challenge has elements that require to link classes to other classes and/or their instances. To avoid redundancy in the definition of such relations, we introduce the following extension to DeepTelos:

```
Enumeration in Class with
  attribute
     member: Proposition;
     memberset: Proposition
  rule
     rmember: $ forall x/Proposition
                       EN/Enumeration
          (EN member x) ==> (x in EN) $;
     rmemberset: $ forall x,S/Proposition
                          EN/Enumeration
        (EN memberset S) and (x in S)
          ==> (x in EN) $
end
```

The class allows defining enumerations of explicit instances ("member") and instances of classes ("memberset").

2.2.2 Using DeepTelos in ConceptBase

To motivate the use of DeepTelos, consider the following simple scenario. A car has a model number and a mileage. In potency-based approaches to multi-level modeling (Atkinson et al. 2009; Atkinson and Kühne 2001; Frank 2014; Lara et al. 2014), one would have a meta class "CarModel" (M2 level) with two attributes. The attribute "model number" would be applicable to instances of "Car", i.e. classes at M1 level (potency 1). The attribute "mileage" would be applicable to

instances of instances of "Car" (M0 level, potency 2). In DeepTelos, the potency levels are replaced by most-general instances:

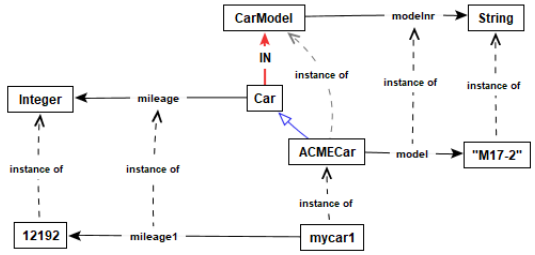

Figure 2: DeepTelos solution for the car example.

The object "CarModel" is a meta class with the most-general instance "Car". The model number is applicable to instances of "CarModel", whereas the mileage is applicable to instances of "Car". In the above example, "ACMECar" is defined as an explicit (Telos) subclass of "Car". By DeepTelos rule "mrule4" (Sect. 2.2 and sources at http://conceptbase.cc/emisaj2021challenge), it is then a derived instance[4] of "CarModel" and thus may use the "modelnr" attribute. The object "mycar1" is then an instance of "ACMECar" (and thus of "Car") and can instantiate the mileage attribute. The concepts "Car" and "CarModel" shall be regarded as one aggregated concept: "Car" is the proxy of "CarModel" at the abstraction level below "CarModel". Thus, the purpose of the most-general instance "Car" is to be able to define attributes like mileage that are applicable to all cars.

As a second example, consider the application of most-general instances to the software process model of Fig. 1. Figure 3 virtually duplicates the structure of the metaclass level to the class level. Consider the most-general instances "DesignObject", "DesignDecision", "DesignTool" and their relations "from", "to" and "by" as proxies of their counterparts at the metaclass level. They make the predicate `(x from y)`, `(x to y)` and `(x by y)` available to instances of instances of "DesignDecisionType". At first glance, this appears like redundancy. The deductive rules defining DeepTelos derive the specialization facts

```
(TDL_EC_DO ISA DesignObject),
(DBLP_REL_DO ISA DesignObject),
(EntHierMapMoveDown ISA DesignDecision),
(MappingAssistant ISA DesignTool).
```

The objects at the bottom are thus instances of these proxy classes and can be queried as their instances. The example shows that relations such as the "from" relation of "DesignDecisionType" can also have most-general instances in DeepTelos. In the textual syntax, this is expressed as

```
DesignDecision!from with
  IN class: DesignDecisionType!from
end
```

The inclusion of the constructs for "DesignObject", "DesignDecision", "DesignTool" and their relationships allows one to query objects like "mapInvitations" via the most-general instances. For example, one can list all instances of "DesignDecision" that are linked to the design object "Papers", without having to know the explicit class "TDL_EC_DO" of "Papers". Note that the model is not strictly layered into disjoint abstraction levels. For example, the object "mapInvitations" is at the lowest abstraction level, but the object "Papers" is arguably a class. This is typically for process models where the result of activities (here design decisions) are software artifacts, such as classes.[5] The most-general instances also allow one to define attributes such as "vendor" for "DesignTool". Then, the instance "MA_exec1" can directly instantiate this attribute without having to define it for its explicit class "MappingAssistant".

2.2.3 Modules in ConceptBase

For supporting model reuse, we employ the module system of ConceptBase to organize the model

[4] Derived links are visualized by gray arrows in the diagrams. This applies to derived instantiations and to derived specializations. Explicit specializations are indicated by blue arrows. All diagrams in this paper are created with ConceptBase from the original models.

[5] This level-mismatch was also observed by Gonzalez-Perez and Henderson-Sellers (2006). It highlights why pure UML-based approaches are limited when it comes to process modeling.

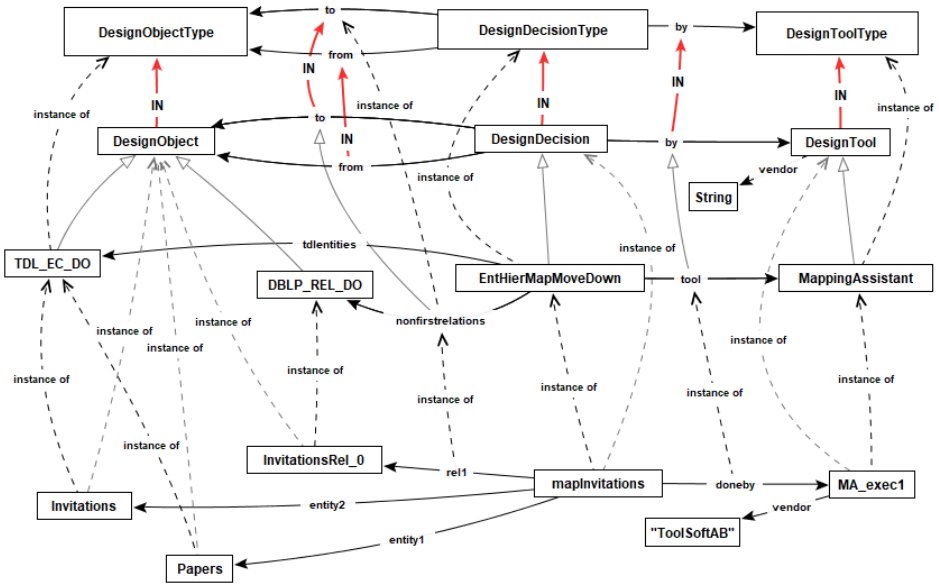

Figure 3: Multi-level version of the software process model adapted from Jarke et al. (1990, p. 98).

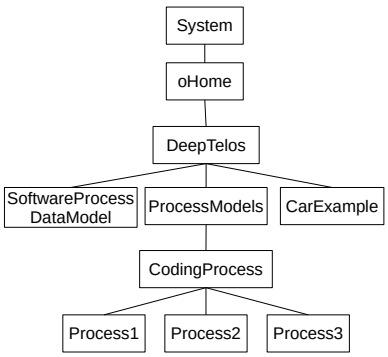

Figure 4: ConceptBase modules used in this paper.

definitions and separate variants of models. ConceptBase supports sub-module hierarchies, where a sub-module "sees" all definitions made in the super-module but not the definitions inside sibling modules. There are two predefined modules in ConceptBase: the root module "System" contains all builtin objects of Telos and ConceptBase, in particular the omega class "Proposition". The "System" module has a sub-module "oHome", which contains the definitions made by a user of ConceptBase.

We shall use the "oHome" module for the definition of some useful formulas such as cardinality constraints, transitivity, symmetry and other frequently used relational properties. Inside the "oHome" module, we define the module DeepTelos, which includes the 5 rules and the one constraint discussed above plus some graphical types for a nicer visualization of DeepTelos most-general instance hierarchies. Then, the DeepTelos module contains a sub-module "ProcessModels", which contains the solution for the requirements P1-P19 of the challenge, i.e. the constructs for defining process models. The sub-module "CodingProcess" of "ProcessModels" contains the example software engineering process model of the multi-level process challenge. Finally, the sub-modules "Process1", "Process2", and "Process3" of "CodingProcess" contain example processes of the process type defined in "CodingProcess". The car example of Fig. 2 is stored in another sub-module of DeepTelos, sharing the definitions of DeepTelos, "oHome" and "System" but not of "ProcessModels" and its sub modules. The legacy software process data model is managed in the sub-module with the same name.

The sources for the modules in Fig. 4 are provided via the link in Sect. 10.1 at the end of this paper. In the next section, the multi-level process challenge is outlined and key concepts are assigned to their designated abstraction levels.

3 Analysis of the multi-level process challenge

This section discusses the abstraction levels of concepts mentioned in the multi-level process challenge. The challenge defines three main concepts. First, there are *tasks* and task types. Task types are used to define process models (roughly M1 level). The process models can be instantiated and deliver processes (M0 level). A process is the trace of the execution of a process model for a given case (e.g. to develop a software system). The second main concept is the *actor* type, resp. actor. Actor types are related to task types, e.g. to define the required competence of an actor to execute a task type. Thirdly, there are *artifacts* and artifact types. They define the inputs and outputs of task types (M1 level) and of their instances (M0 level). At the M1 level, task types define the types of inputs and outputs whereas the M0 level defines which actual input and output artifacts were used in a specific execution of the process, resp. its tasks. Some attributes and relations link objects at the same instantiation level, while others link objects at different abstraction levels. For example, one can authorize a single actor like "AnnSmith" (M0) as the sole person allowed to execute a given task type such as "CodingInCobol". A particular such cross-level attribute is the "lastupdate" attribute (requirement P19 of the challenge). Our solution will allow to specify the last update time for any concept at any instantiation level.

While DeepTelos does not have predefined levels such as M0, M1, M2, M3 and so forth, it is still useful to roughly identify objects that an OMG-educated modeler would assign objects to.

M0: Here we find objects like "AnnSmith", task instances such as "coding1" as instance of "CodingInCobol", which is started at a given date and ends at another date. We also find artifact instances such as "cobolprogram1". It should be noted that such artifacts can contain objects at a higher instantiation level. For example, a design document can contain a whole UML class diagram (M1) level. We have discussed this phenomenon earlier in the context of process-data diagrams (Jeusfeld 2011, pp. 3–5). The solution there did however not use DeepTelos, but declared certain objects to be both instance and specialization of meta class.

M1: This level contains the specification of a process type such as the ACME coding process type of the challenge. This level roughly corresponds to a BPMN process model. In contrast to BPMN, our solution (and the challenge) also covers the execution of the process model at M0. In our solution, the concepts "Task", "Artifact", "Actor", and "Process" are all at the M1 level.

M2: The M2 level defines constructs such as "ArtifactType" (having the most-general instance "Artifact"), "TaskType" (most-general instance "Task"), "ActorType" (most-general instance "Actor"), and "ProcessType" (most-general instance "Process"). Certain subclasses of these classes are also defined at this level such as "CriticalTaskType". We re-use the definition of a core BPMN language at the M2 level and pull it from the ConceptBase Forum.[6] Such reuse of existing metamodels simplifies the solution to the challenge.

M3: The M3 level contains the meta-meta classes "Node", "NodeOrLink" and the link object (label "connectedTo"). We reuse these definitions since the core BPMN language was defined with these meta-meta classes.

Omega: We make heavy use of the omega-class "Proposition" as discussed in the introduction to define DeepTelos and to provide the "lastupdate" attribute. The omega level also contains the implementations of multiplicity constraints such as "necessary" (1..*) and single (0..1), see also the sources at the web page http://conceptbase.cc/emisaj2021challenge/SOURCES.

[6] http://conceptbase.cc/CB-Forum.html

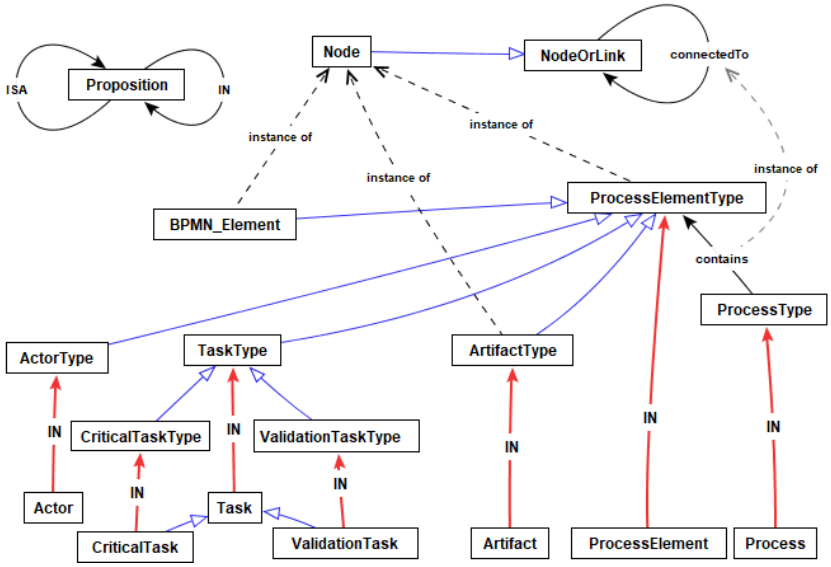

Figure 5: Concepts for multi-level process modeling.

The challenge demands a particular property of task types, namely the planned duration. In our solution, we shall allow to compare the planned duration (task type) with the actual duration of a task instance (derived from start and finish date). Function definitions are used for that purpose. We also define a number of queries to analyze process models and their execution.

4 Model presentation: The constructs

This chapter introduces the solution of the process challenge by DeepTelos and ConceptBase. The figures are also included on the web page mentioned in Sect. 10.1, where they are linked to executable graph files to be processed by the ConceptBase graph editor.

4.1 The levels

Fig. 5 shows the abstraction levels used for the solution. On the top left is the object "Proposition" defining the two DeepTelos relations "IN" (for relating a most-general instance to its class) and "ISA" (for derived specialization relations). These two relations extend the so-called omega-level of Telos. All explicit objects in this diagram including all nodes and links are instances of the omega-class "Proposition". The next level is established by the objects "Node", "NodeOrLink" and the "connectedTo" link of "NodeOrLink". In traditional metamodeling, these objects would be dedicated as metameta classes, i.e. members of the M3 or even M4 level. It is used to define modeling languages such as BPMN. In the figure, the object "BPMN_Element" is the superclass of all constructs of a core BPMN metamodel that we utilize in our solution. Since this metamodel does not cater for all constructs needed for the challenge, we also define a meta class "ProcessElementType", which subsumes all required constructs including the core BPMN constructs. All subclasses of "ProcessElementType" plus the class "ProcessType" would be regarded as M2 level in a UML environment.

The next level is formed by the most-general instances "Actor" (of "ActorType"), "Task" (of "TaskType"), "Artifact" (of "ArtifactType"), "ProcessElement" (of "ProcessElementType") and "Process" (of "ProcessType"). Hence, "Task" is a simple class (regarded as M1 level in UML) and has all instances of instances of "TaskType" as instances, as derived via the DeepTelos rules. The instances of the displayed most-general instances form the

M0 level. We will see such instances in the section for the example processes. In summary, the solution features four UML-ish abstraction levels plus the omega level.

4.2 Requirements P1-P3

- P1: A process type (such as "claim handling") is defined by the composition of one or more task types (receive claim, assess claim, pay premium) 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.
- 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).

Fig. 6 shows the solution to requirements P1-P3. "ProcessType" is modeled as a container for "ProcessElementType", which subsumes all required constructs (including gateways, start/end elements and task types). The ordering of the process element types is supported by the "next" relation of "BPMN_Element". The two subclasses "PlaceLike" and "TransitionLike" are used to embed BPMN into a Petri-net semantics, which we do not use in this solution but can be inspected in the publicly available models given in the footnote.[7] The figure also shows the object "Process" as most-general instance of "ProcessType". It features a "contains" relation, which is the most-general instance of the corresponding relation of "ProcessType". Hence, the "contains" relation at the M2 level is propagated to the M1 level. Just as we define process models as containers of process element types, we define processes as containers of process elements.

The complete source code is available via the link in Sect. 10.1.

[7] http://conceptbase.cc/dynamic-models.html

4.3 Requirements P4-P6

- P4: Each task type is created by an actor, who will not necessarily perform it. For example, Ben Boss created the task type "assess claim".
- 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 (e.g., John Smith and Paul Alter may be the only actors who are allowed to assess claims).

These requirements introduce the first cross-level relations between "Actor" (M1) and "TaskType" (M2).

The relation "creator" links a task type to the actor, who created it. Figure 7 also shows an instance of this relation (see link to actor "BobBrown"). The relation "executortype" links "TaskType" and "ActorType" (requirement P5). The executor type of a task type can also be an enumeration (see Sect. 2.2.1). This addresses requirement P6. The definition of "CobolCoder" in DeepTelos is:

```
CobolCoder in Enumeration isA Developer with
  member m1: AnnSmith end
```

The label "m1" here is distinguishing several entries under the attribute category "member". This lifts the object "AnnSmith" to the singleton class "CobolCoder".

Fig. 7 shows two links that are derived by rules. First, "TestCaseDesign" is an explicit instance of "BPMN_Activity", which itself is an explicit subclass of "TaskType". Then, by a predefined rule of specialization, "TestCaseDesign" is a derived instance of "TaskType". Second, since "TestCaseDesign" is an instance of "TaskType", the DeepTelos rule "mrule1" derives that "TestCaseDesign" is a subclass of "Task". The derived links are shown here for explaining the function of the Telos and DeepTelos constructs. They are not typically included in diagrams created by modelers.

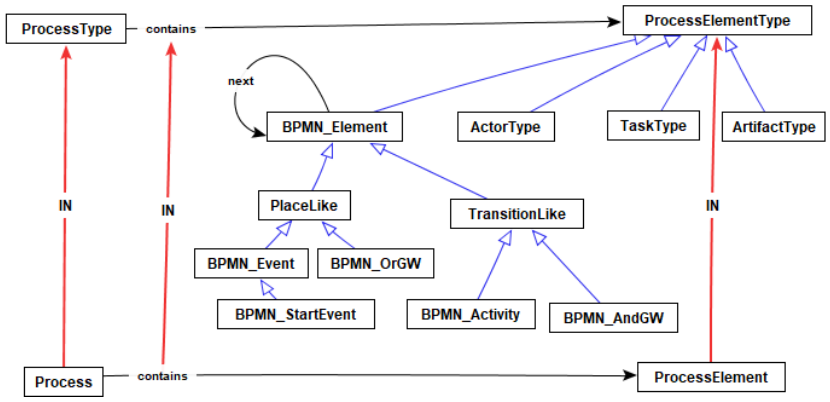

Figure 6: Requirements P1-P3.

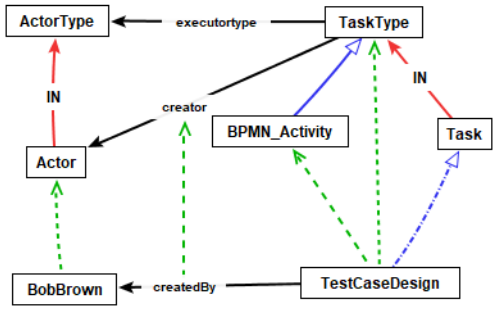

Figure 7: Requirements P4-P6.

4.4 Requirements P7-P9

- P7: For each task type (such as "authorize payment") 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.

Task types use and produce artifact types. This is also propagated to tasks, which use and produce artifacts.

Critical task types and validation task types are modeled as subclasses of "TaskType". The query class "CriticalButNotValidatedM1" checks requirement P9. This query returns all critical task types that are not checked by a validation task type. A similar query "CriticalButNotValidatedM0" is defined for the most-general instance "CriticalTask". The code for both query classes is available via the source code link in Sect. 10.1.

The latter query operates at the M0 level, i.e. checks actual executions of the process rather than the process type. In addition, constraints (see class "CriticalTaskType" in the source code via 10.1) are used to address the actor requirements of P9.

Finally, Fig. 8 shows the planned duration (requirement P8) and the actual duration (property of Task). The latter is not demanded by the challenge but we found it useful to later check whether a task is delayed.

4.5 Requirements P10-P16

- P10: Each process type may be enacted multiple times.
- P11: Each process comprises one or more tasks.
- P12: Each task has a begin date and end date.
- P13: Tasks are associated with artifacts used and 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 (e.g., Senior Manager and Project Leader).
- P16: An artifact may have more than one artifact type.

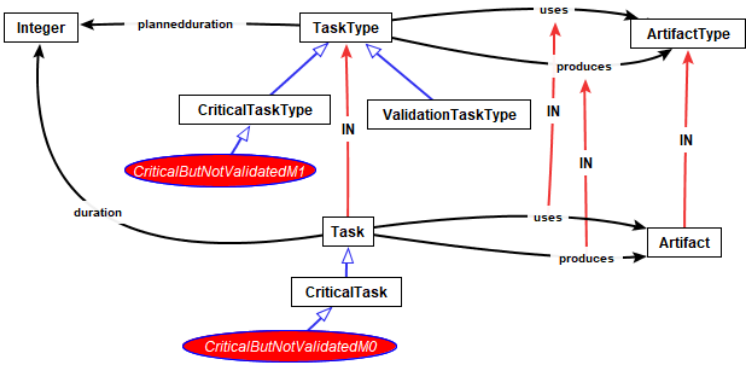

Figure 8: Requirements P7-P9.

Requirement P10 is automatically fulfilled by Telos since any process type (instance of "ProcessType") may itself have any number of instances. We shall later provide instances of the ACME example process type. Requirement P11 is fulfilled by the "contains" relation of "Process", see Fig. 6. "Task" is a subclass of "ProcessElement". For requirement P13, we refer to Fig. 8. Of particular interest in this solution is that the "uses" and "produces" relations for "Task" are most-general instances of their counterparts defined for "TaskType". As a consequence, an instance of "TaskType" such as "Testing" (see Fig. 13) would instantiate the "uses" and "produces" relations defined for "TaskType". These relation instances are then subclasses of the "uses" and "produces" relations for the most-general instance "Task". This allows querying instances of "Task" using the generic relation names "uses" and "produces". Requirement P12 (begin and end of a task) is implemented by two attributes "begindate" and "enddate" of "Task":

```
Task in Class with IN  class: TaskType
  attribute
    uses : Artifact; produces : Artifact;
    begindate : Integer; enddate : Integer;
    duration : Integer; executor : Actor
end
```

The full definition is in the source code via Sect. 10.1. It calculates the actual duration of a task by a deductive rule based on the function "taskDuration". Since the task type of a task has a "plannedduration", we can retrieve delayed tasks by a query class "DelayedTask":

```
DelayedTask in QueryClass isA Task with
   constraint isDelayed:
      $ exists T/TaskType pd,d/Integer
      (this in T) and (T plannedduration pd)
      and (this duration d) and (d > pd) $
end
```

This approach also allows defining a derived attribute ‘avgduration’ of TaskType, which is the average of all duration values of its instances (not implemented here). This supports datawarehouse-like aggregation of class-level attributes from instance-level attributes.

Requirement P14 is realized by the query class "UnmatchedTask" (see Sect. 10.1). Requirements P15 and P16 is also fulfilled by Telos since each object may have multiple classes. Hence, any actor can have multiple actor types. A similar argument holds for requirement P16. The processes "Process1" to "Process3" have examples for such objects that have multiple classes.

4.6 Requirements P17-P18

- 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.

Requirement P17 demands that each actor who performs (=executes) a task must be authorized to do so. The requirement is related to the "executortype" relation of "TaskType". Our solution to this requirement does not implement defaults when authorizations are not defined explicitly. Default rules could however be added quite handily since Datalog realizes negation as failure.

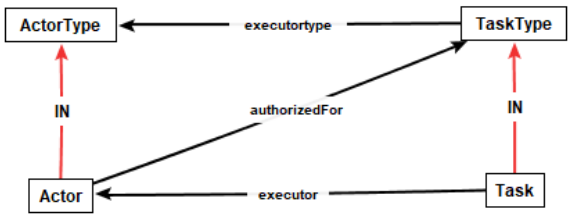

Figure 9: Requirement P17.

Fig. 9 shows the relevant defined relations. The authorization is then expressed by a Telos constraint "isAuthorized" of class "Actor":

```
Actor in Class with
   attribute authorizedFor: TaskType
    constraint
    isAuthorized :
 $ forall a/Actor t/Task T/AuthorizedTaskType
         (t in T) and (t executor a)
         ==> (a authorizedFor T) $
end
```

The query class "AuthorizedTaskType" is used to filter out those tasks that have no authorization declared. The solution is only about half of what should be expected by a proper authorization schema. We argue that it could be extended but foresee a rather complicated specification by Telos rules and constraints involving negated predicates to model defaults. Requirement P18 is again easily fulfilled by Telos. All instances of sub-classes are also instances of super-classes in Telos. Consequently, rules and constraints applicable to instances of the super-classes also apply to instances of the sub-classes.

4.7 Requirement P19

- P19: All modeling elements, at all levels, must have a last updated value of type time stamp.

This requirement demands that all objects and classes in a model should have an attribute "lastupdate" to check when it has been modified for the last time. ConceptBase is a temporal database storing the transaction time interval with each proposition. This allows a generic solution to this requirement since all updates to objects are realized by inserting (start of transaction time) and deleting (end of transaction time) propositions. The rule to derive the "lastupdate" attribute is:

```
forall o/Proposition tt/TransactionTime
 (tt = lastUpdateTime(o)) ==> (o lastupdate tt)
```

It uses a function "lastUpdateTime", which is available via the source code link in Sect. 10.1. The solution applies to all objects ("propositions"), not just DeepTelos objects. Since the transaction time of an object is itself an object, one could also store the name of the user who performed the transaction. By this, one can even check who has created what model elements at which time down to the granularity of single attributes. An example of the derived attribute "lastupdate" is shown in Fig. 18. The solution is realized by a single function definition plus the above generic rule applicable to any object in the database regardless of its abstraction level.

5 Model presentation: example process

The example ACME software engineering process of the challenge is modeled using the capabilities of the core BPMN language implemented by a Telos metamodel:

The representation of the sequencing between the start element, the task types, the gateways, and the end element is done by using the ‘next’ relation of BPMN_Element (see Fig. 6). The task type "CodingInCobol" is defined as a subclass of "Coding". This is used for a variant of the process type. Fig. 10 uses BPMN-style graphical shapes where appropriate.

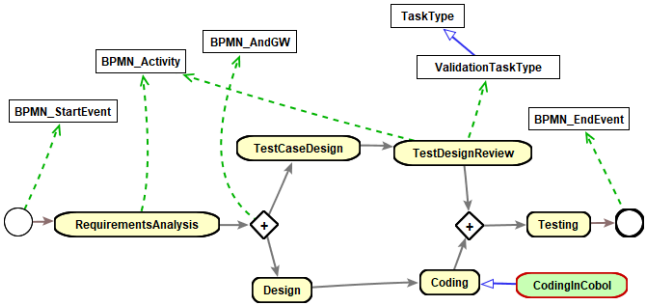

Figure 10: The ACME process type in our solution.

In the full solution, the BPMN constructs are mapped to an executable Petri net, i.e. enabled tasks can be fired in a kind of a "token game". One can analyze certain well-formedness properties such as that each task must be on a path from a start event to an end event. Modeling errors like joining a parallel "and" gateway by an "or" gateway can be detected by appropriate query classes. This is however beyond the scope of this paper.

5.1 Requirements S1-S4

- 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.

The upper part of Fig. 11 shows the constructs for "TaskType", "ActorType" and "ArtifactType" (M2 level) . The executor of the task is assigned using the "executortype" relation of "TaskType". The produced artifact type, e.g. "TestCaseDocument" (M1 level) uses the "produces" relation of "TaskType". The artifact type "ProgramCode" has a "language" attribute. A similar attribute is defined for the "Coding" task type.

To understand the instantiations derived by Telos, consider the object "DeveloperOrTestDesigner". It is defined as an (explicit) instance of "NormalActorType", which is an explicit specialization of "ActorType". Via DeepTelos rule "mrule4" (see Sect. 2.2) it is then also an instance of "ActorType". This instantiation then allows using the "executortype" relation for linking TestCaseDesign to its executor "DeveloperOrTestDesigner". Note that the figure does not show all instantiations to keep it readable.

5.2 Requirements S5-S7

- 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.

These requirements demand that the task type "CodingInCobol" always produces Cobol code and that Cobol is the languages used in this task type. Moreover, Ann Smith is a developer and the only person authorized to execute this task type.

In the DeepTelos solution, the "Coding" process produces "ProgramCode" (relation artifact1). This is specialized to "CobolCode" for "CodingInCobol". Ann Smith is the actor developer, who can execute "CodingInCobol". The type "CobolCoder" is defined as an enumeration, as discussed in the solution to requirements P4-P6. The enumeration type lifts the instance "AnnSmith" to the singleton type "CobolCoder", which has just one instance. The instantiation of "AnnSmith" to "CobolCoder" is derived via a generic rule of "Enumeration". The language Cobol is prescribed to "CodingInCobol" by a constraint:

```
CodingInCobol in BPMN_Activity isA Coding with
   constraint useCobol:
    $ forall cic/CodingInCobol
      (cic useslanguage Cobol) $
  executortype coder : CobolCoder
  produces artifact1 : CobolCode
end


CobolCoder in Enumeration
   isA Developer with
   member m1 : AnnSmith
end
```

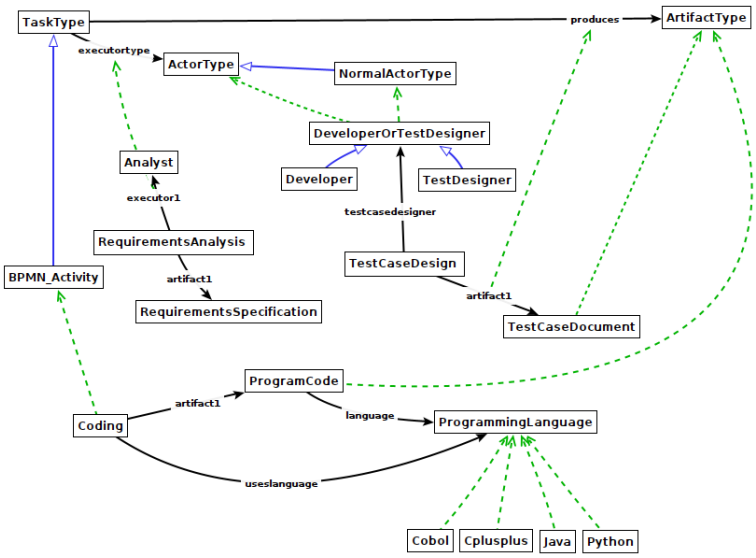

Figure 11: Requirements S1-S4.

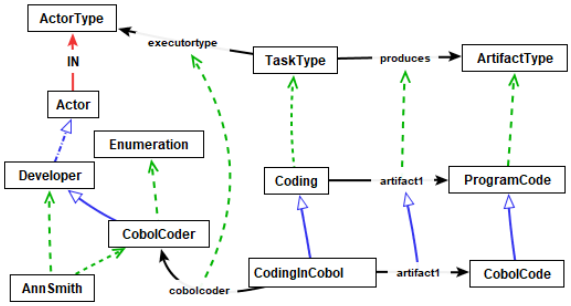

Figure 12: Requirements S5-S7.

Note that "CobolCoder" is also a subclass of "Actor", hence via the axioms of DeepTelos it is an instance of "ActorType". This makes the association "executortype" an allowed class for the "cobolcoder" link. Thus, the "executortype" of "CodingInCobol" is "CobolCoder", and "Ann-Smith" is the only instance of that class.

5.3 Requirements S8-S10

- 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.

The "Testing" task type shall be performed by a "Tester" and it shall produce a "TestReport" as artifact. A test report is associated to other software engineering artifacts produced by other tasks.

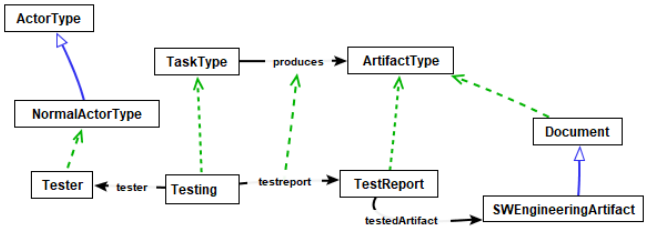

Figure 13: Requirements S8-S9.

In the DeepTelos solution, "SWEngineeringArtifact" is the superclass of all artifact types produced by the ACME process. It is a subclass of "Document", which is an instance of "ArtifactType". The DeepTelos rules then derive that "SWEngineeringArtifact" is also an instance of "ArtifactType".

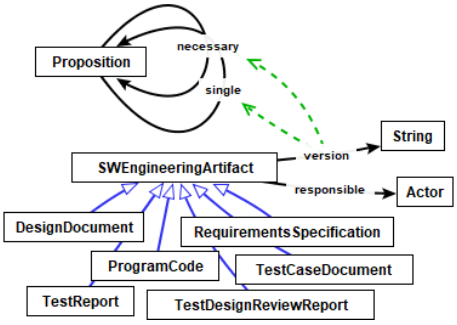

Figure 14: Requirement S10.

Fig. 14 shows the solution to requirement S10. The latter is defined to be necessary (1..*) and single-valued (0..1). These two cardinalities are defined by appropriate multi-level formulas in the oHome module as discussed in the introduction.

5.4 Requirements S11-S12

- S11: Bob Brown is an analyst and tester. He has created all task types in this software development process.
- S12: The expected duration of testing is 9 days.

The actor Bob Brown is a tester and analyst. He is also the creator of all ACME task types. Furthermore, the expected duration of testing is 9 days.

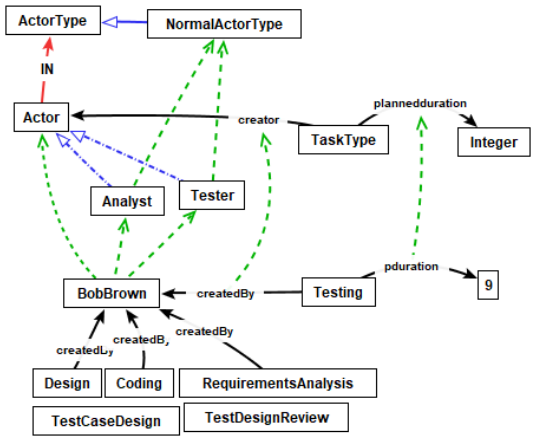

Figure 15: Requirements S11-S12.

Again, the DeepTelos rules take care of the necessary derived instantiations and specializations: "Analyst" and "Tester" are both instances of "NormalActorType", which is a specialization of "ActorType". Then, both "Analyst" and "Tester" are derived instances of "ActorType", and consequently "Analyst" and "Tester" are derived specializations of "Actor". Then, "BobBrown" becomes an instance of "Actor" as well and can instantiate the "createdby" attribute of "TaskType". "BobBrown" is associated as creator of all ACME task types.

5.5 Requirements S13

- 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.

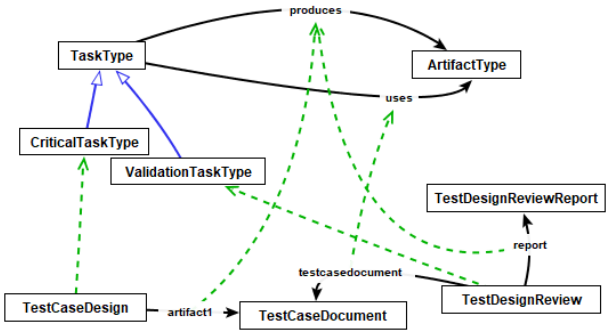

Figure 16: Requirement S13.

The test case document is produced by "TestCaseDesign" (a critical task) and used by "TestDesignReview" (a validation task). The requirement that the executor must be a senior analyst is addressed by a constraint of class "TestCaseDesign":

```
forall tcd/Task a/Actor (tcd in TestCaseDesign)
  and (tcd executor a) ==> (a in SeniorAnalyst)
```

The definition of "TestCaseDesign" plus all of the source code and examples are available via the weblink specified in Sect. 10.1. This completes the discussion of the ACME case requirements. Note that the definitions are mostly at the M1 level (simple classes). Actual executions of the ACME process type deliver instances of the tasks, i.e. objects at the M0 level. We provide three submodules "Process1", "Process2", and "Process3"

for such examples. The first is an example of the standard ACME process, the second uses the variant with "CodingInCobol", and the third adds contents to the software engineering artifacts such as actual lines of code to the instances "CobolCode". The third submodule "Process3" shows how to trace dependencies between artifacts including dependencies between individual model elements, e.g. between specific lines of code and specific requirements.

The diagram in Fig. 17 displays mostly explicit facts. There are two exceptions. First, the two "duration" links (indicated by the broken black lines) are derived by a simple rule subtracting the start/end attributes of a task (labels "d1" and "d2"). Second, the two instantiation links to the class "DelayedTask" (which is a so-called query class) are derived by the constraint of "DelayedTask", see requirements P12. The two instantiation links have slightly more dot than the explicit instantiations in the figure.

6 Example process traces

Instances of tasks form traces of the executions of process types such as the ACME process. Actual instances of tasks have a begin date, a finish date, and are performed by suitable actors. They also produce and use instances of the artifact types specified in the ACME process type. The process instances are at the lowest abstraction level (M0), though artifacts may actually contain models (compare Jeusfeld 2011) such as a UML class diagram.

The example process trace in Fig. 17 shows a violation of the pattern "critical task (testcasedesign1) not validated".

Fig. 18 shows transitive dependencies between model elements (here code lines via design elements to requirements). Model traceability was not included in the list of requirements of the challenge. However, it comes as an inexpensive bonus by a small set of rules. One rule derived a direct dependency between artifacts, if there is a task that produces an artifact from a used artifact. Secondly, this direct dependency is made transitive by a transitive closure rule. The precise solution is included in the sources accessible via the web page listed at the end of the paper (see Sect. 10.1).

7 Discussion

DeepTelos is a straightforward extension of Telos and defined by just 5 deductive rules interpreted by a Datalog engine.

7.1 Employed levels

DeepTelos has no level numbers and no potencies. Instead, levels are introduced by declaring a relation like (Task IN TaskType). So, the instances of "TaskType" form one level below "TaskType", and the instances of "Task" form another level below "Task". The main levels of this solution are shown in Fig. 5. As discussed earlier, one can identify the 4 UML-ish levels M0 to M3 in our solution plus the omega level ("Proposition"). In this solution, we had no chain of most-general instances such as (m1 IN m2), (m2 in m3). This indicates that there are only potencies 1 and 2 used for attributes in our solution. Consider the position of "Node" in Fig. 5 as a level on top of "ProcessElementType" to create such a chain. DeepTelos spans levels by the (m IN c) predicate, but they are existing in parallel and have no static level number[8] .

7.2 Cross-level relationships and cross-level constraints

Such relationships were always possible in Telos. DeepTelos makes this feature even more useful, since such relations can be defined between objects that stand in most-general instance relation. For example, the object "TaskType" has a relation "createdBy" to "Actor". "Actor" is a most-general instance of "ActorType" and is related to "TaskType" by other (same-level) relations. As mentioned earlier, we have formally no static level numbers. Instead, we (intuitively) derive

[8] As shown in http://conceptbase.cc/deeptelos, one can define objects: (M0Object IN M1Object), (M1Object IN M2Object), (M2Object IN M3Object) to force objects into UML-ish levels. There is however no apparent advantage to do so.

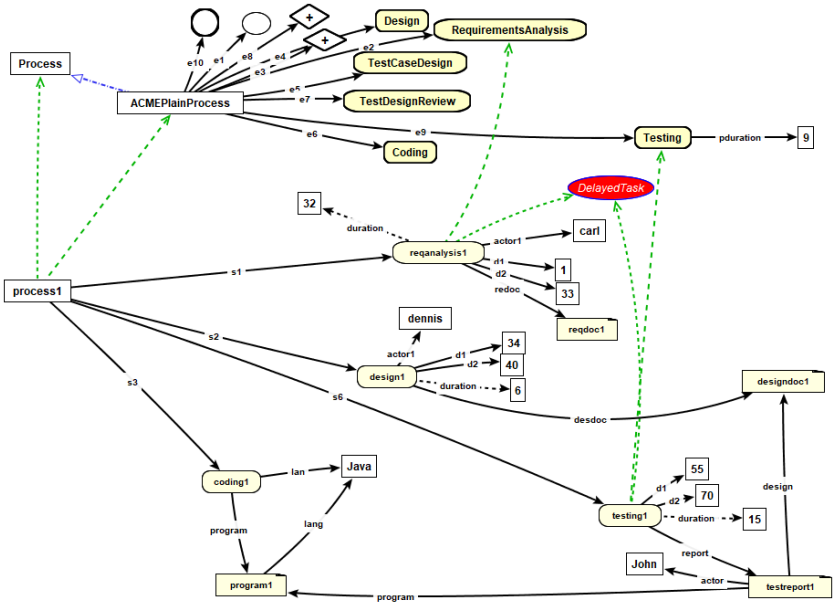

Figure 17: Computing delayed tasks.

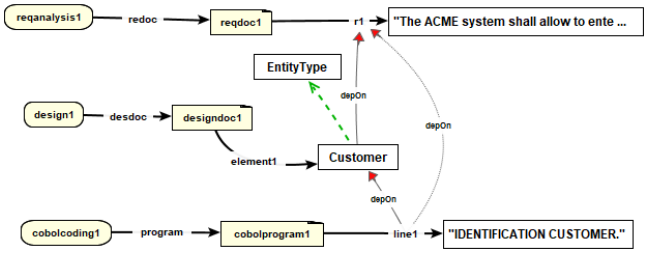

Figure 18: Transitive dependencies between artifacts.

the numeric level of an object by the chain of instantiations.

For example, "BobBrown" has no instance but is instance of "Actor", "Actor" is (most-general) instance of "ActorType", which is an instance of "Node". Hence, we would associate "BobBrown" to level M0, "Actor" to M1, "ActorType" to M2, and "Node" to M3. In general such a calculation is not delivering unique level numbers in DeepTelos. But the intuition is still useful. Orthogonal to all these levels is the omega level: objects of any level are also instances of "Proposition". One may interpret this level as the linguistic level of Telos.

Cross-level constraints are also used in our solution, e.g. to define the "authorizedFor" relation as solution of requirement P17 (see Sect. 4.6):

```
forall a/Actor t/Task T/AuthorizedTaskType
     (t in T) and (t executor a)
==> (a authorizedFor T)
```

The variable *a* ranges over the "Actor" level (M1) and *T* over the "TaskType" level (M2). The relation "authorizedFor" is also a cross-level relationship. Since Telos regards all explicit information as objects, such constraints are technically no different from other constraints.

7.3 Integrity mechanism

Integrity constraints are specified as shown above as first-order formulas, which are compiled to Datalog rules using the algorithm developed by Lloyd and Topor (1984). Datalog defines the minimal Herbrand model of a Datalog program via a fixpoint engine. ConceptBase includes such an engine. Technically, the integrity constraints are formally negated to derive the violation of the constraint ("denial form"). If the violation can be derived, the corresponding update to the database (i.e. Telos models) is rejected. The Datalog engine uses the closed-world assumption (CWA)

for handling negated predicates. This yields a fast implementation but is not equivalent to classical negation. However, the advantage of the CWA is that it naturally computes the transitive closure of relations. Rules defining transitive closures play an important role in (software process) modeling, e.g. to realize traceability.

For example, dependencies between artifacts can be traced via a transitive closure. In this solution we also used queries to formulate constraints, see "CriticalBut-NotValidatedM0/M1" in Sect. 2. Queries cannot be violated but can be used to return violators as their answer. This is in many cases the preferred way in ConceptBase to define constraints. It allows working with incomplete models that technically violate some constraints while being completed. A formal integrity constraint would forbid such models. A query can be called at any time and the modeler can then change the models to reduce the number of violators.

Cardinality constraints are defined via the "necessary", "single" categories (compare Fig. 14). The constraints are maintained by ConceptBase. A violation leads to the rollback of the model update that first introduced the violation. Cardinality constraints can be formulated at any abstraction level since ConceptBase represents all explicit model elements as objects. For example, classes and metaclasses are all objects (instances of "Proposition").

7.4 Abstraction

The solution is organized in modules. The upper-level modules have more abstract definitions than the lower level models. For example, the "DeepTelos" module defines the DeepTelos constructs. They can be used in any multi-level modeling project (e.g. the "CarExample"). The "DeepTelos" module is a highly re-usable and not domain-dependent. The "ProcessModels" module contains the definition of processes (tasks, actors, artifacts, etc.). This can be re-used to the extent to which this conceptualization of processes is deemed generic enough. In our solution, we integrated an existing BPMN meta model into this module to save modeling time.

In the original BPMN meta model, a Petri net like semantics was inherited by the BPMN meta model: tasks could be triggered and that led to a token flow to the next task. We used an active rule (ECA) to implement the token flow semantics. However, this is disabled in this solution because the processes of the challenge have more of a "colored Petri net" semantics. The task instances (M1) could be produced by triggering an active rule that realizes the semantics of the corresponding task type (setting dates, input/output artifacts, executor, ...). Since this cannot be automated due to missing data, we abstained from realizing such dynamic semantics. Still, the "ProcessModels" module is abstract enough to be reused for other process modeling domains. The module "CodingProcess" defines the ACME process example. This is apparently not very reusable in other domains since it is a toy example.

7.5 Deep characterization

Consider as example the two levels (Task IN TaskType). The two levels are closely related via the DeepTelos rules. In particular each instance of "TaskType" becomes a subclass of "Task". As mentioned earlier, DeepTelos replaces potencies by most-general instances. So, a property such as "duration" is defined at the object "Task", not "TaskType". One could see the pair "Task+TaskType" as a single abstract entity. Then, the attribute "duration" is indeed defined at an abstract level and characterizes objects at the UML-ish M0 level.

Another aspect of deep characterization are the multi-level rules used in this solution. They range over three or more instantiation levels and are partially evaluated to sets of rules ranging over just two levels. For example, the "necessary" construct defined the "1..*" multiplicity by a single formula. It characterizes all uses of the necessary construct.

Note that DeepTelos itself is defined by multi-level rules. A multi-level rule is a rule whose labels and variables refer to objects of more than two abstraction levels. For example, in "mrule1" the label "Proposition" refers to an object at the omega

level. The variable "c" refers to an instance of "Proposition", and the variable "x" to an instance of "c". Hence, this rule refers to objects at three abstraction levels. Multi-level rules are partially evaluated by ConceptBase into a set of equivalent two-level rules. The main reason for this partial evaluation is the efficiency of rule evaluation. Another benefit is the ability to extract the specific set of two-level rules that govern a given multi-level model. This may support the mapping of multi-level models into a set of two-level models. This is however not yet supported by our system.

7.6 Reuse

This aspect was already discussed in the previous paragraphs. The reuse is supported by the module structure. The source code of the modules (Telos sources) is shared at http://conceptbase.cc/emisaj2021challenge/SOURCES.

A module source can be directly inserted into a ConceptBase database and then be used. The sources are compiled to a set of facts/objects plus a set of (executable) rules. One can regard them as logical theories. Like with logical theories, one can combine them provided that integrity constraints are satisfied by the combination. In the case of this challenge, we also reused existing Telos sources (BPMN) and integrated them into the DeepTelos concepts. The formalization of BPMN in Telos was done prior to this challenge and included a mapping to Petri nets including their semantics (firing enabled transitions). This semantics is thus also inherited in our solution, i.e. one can evaluate the BPMN process models in our solution by firing "enabled" tasks.

7.7 Semantics

ConceptBase uses a Datalog-neg engine to evaluate rules and constraints. The axioms for the basic constructs for specialization (c isA d), attribution/relations (x m y), and instantiation (x in c) are also expressed as rules and constraints. The semantics of a model in ConceptBase is the minimal fixpoint interpretation (=extension) for the logical predicates occurring in rules and constraints, as computed by the Datalog engine. The DeepTelos rules "mrule1" to "mrule5" (Sect. 2.2) are subject to the Datalog engine as well, deriving particular solutions to the DeepTelos specialization (c ISA d) and the Telos instantiation predicate (x in c). Domain-specific rules and constraints are treated in the same way as any rule or constraint in ConceptBase. For example, there are rules for the predicate (a authorizedFor t). The extension specifies which actor is authorized for which task, which is evaluated at the M0 level (here: process traces). The extension is computed by ConceptBase.

The Telos specialization relationship (c isA d) is axiomatized by Jeusfeld (2005). One of the axioms realizes the Nixon diamond pattern for attributes and relations: for any combination of an object x and a class attribute/relation label m, the class c that defines the most specific m is unique. This axiomatization supports *s*ubstitutability: wherever an instance of the superclass d is allowed, an instance of the subclass is also allowed. Note however that Telos does not support class methods. Substitutability is limited to rules, constraints, and queries. The DeepTelos specialization (c ISA d) shall behave like the Telos specialization, though we did not yet transcribe all axioms for (c isA d) to (c ISA d). The only reason to introduce the DeepTelos specialization is an implementation limitation for the Telos specialization in ConceptBase: it does not support user-defined rules for the Telos specialization predicate.

7.8 Incremental updates

ConceptBase supports incremental updates at any instantiation level at any time. The Telos axioms as implemented by ConceptBase (Jeusfeld 2009) assign at least the builtin class "Proposition" to any object. Hence, any object does have at least the class "Proposition" and can use the features of "Proposition" to assign attributes, relations, sub/superclasses, and classes/instances at any time. In fact, one could start to define objects at the UML M0 level first and attach its M1-level classes subsequently. Similarly, the M2-level of M1-level objects could be defined when the M1-level objects are already defined (as instances of Proposition).

In practice, the modeling of the different abstraction levels usually starts with the more abstract levels. But, the more abstract levels can be extended and modified as long as the builtin axioms and user-defined integrity constraints are satisfied. Rules, constraints and queries can also be defined at any time. They can be deleted as well or be replaced by revisions at any time.

7.9 Lessons learned

One challenge with using ConceptBase was that recursive rules needed to be evaluated while an update to a model was processed. ConceptBase originally disabled “tabling” (caching the extension of derived predicates) of the Datalog-engine during updates. That could lead to infinite loops since tabling is essential to avoid infinite loops for recursive predicates. We addressed this loophole by temporarily re-activating tabling during updates. Another lesson learned was that the formula compiler of ConceptBase ignored the specialization facts derived by DeepTelos to check the typing of predicates. This forced us initially to some awkward definitions for some queries and redundant specialization facts. This weakness has been removed.

Finally, the more most-general instances are defined, the more rules and constraints are generated by the partial evaluator. Hence, it may be more efficient not to partially evaluate the DeepTelos rules for very large models.

7.10 Further aspects

The partial evaluator generated about 150 two-level rules from the 6 multi-level rules. The number of generated rules depends on the number of instances and sub-classes associated to objects matching the predicate (m IN c). Still, one could use the two-level rules instead of the DeepTelos rules to carve out sub-sets of the modules, e.g. just the CodingProcess plus its sub-module.

Some additional features were added to the process model, in particular to check delayed tasks. This showcases the ability of ConceptBase to evaluate arithmetic and function expressions.

ConceptBase was originally developed as design repository for data-intensive applications, project DAIDA (Jarke 1993). The metameta model of the design repository had the concepts "DesignDecision" (="Task"), "DesignObject" (="Artifact"), and "DesignTool" (roughly "Actor"). It did also feature all abstraction levels used in the challenge. The main drawback was the missing multi-level modeling aspect. This led to many instantiations of the produces/uses relations, while in our solution, we only need to define it for "TaskType" and "Task". The DAIDA project also pioneered the fine-grain traceability of requirements to code lines. An updated version of this feature is available from Jeusfeld (2009, pp. 160ff).

8 Comparison to MULTI 2019 challenge solutions

The multi-level process challenge was first published for the MULTI 2019 workshop (https://www.wi-inf.uni-duisburg-essen.de/MULTI2019/), where it attracted three solutions using different multi-level modeling tools. In this section, we discuss these competing solutions and highlight the main differences to the solution with DeepTelos.

Somogyi et al. (2019) presented a solution for the process challenge based on the DMLA system. DMLA features as main construct step-wise refinement of class attributes, which is kind of a combination of instantiation (when some attribute slot is getting a value) and specialization (when slots are cloned to the refined class). DMLA is level-agnostic like DeepTelos. DeepTelos however does clearly differentiate instantiation from specialization. Both approaches support cross-level relations (i.e. between classes at different abstraction levels). DMLA checks conformance of objects to the class level by its Bootstrap engine (an abstract state engine). DeepTelos relies on integrity constraints expressed in a first-order logic syntax to perform this function.

Rodriguez and Macias (2019) present a solution to the challenge with the MultEcore system. MultEcore has explicit levels with three-fold potencies for attributes. Cross-level relations are realized by so-called META blocks. Objects may have a secondary type, which can be used to attach attributes applicable to objects of multiple abstraction levels. MultEcore appears to require a certain amount of duplication of attributes. An important advantage of MultEcore is its embedding into the Eclipse Modeling Framework (EMF). Hence, MultEcore is de facto a multi-level extension to the Eclipse modeling ecosystem. DeepTelos allows multiple classification (one object may have any number of classes) and multiple generalization (on class can have several superclasses), which was used to address the challenges of relating objects from different abstraction levels and to assign attributes to objects at any abstraction level (e.g. the creation time of an object).

Frank (2019) presents another level-based solution to the challenge using the FMMLx system. FMMLx is based on the XModeler tool, which realizes the XCore meta model. XModeler has an expressive constraint language extending the object-constraint language (OCL). By this, FMMLX can not only formalize the semantics of the multi-level constructs but can also evaluate method specifications for objects. In contrast to DeepTelos, FMMLx realizes single classification (an object can only have one direct class) and also single generalization. Telos as implemented by ConceptBase offers recursive function definitions to computes the values of certain numeric attributes, e.g. the duration of tasks in a process trace. While FMMLX is conceptually close to object-oriented programming languages, DeepTelos follow more the knowledge representation/database tradition.

This paper is an extension to the MULTI 2019 challenge solution with DeepTelos (Jeusfeld 2019). The main technical difference is that the new solution now utilizes enumeration classes to avoid a certain redundancy of the MULTI 2019 with respect to the "executortype" feature (requirements P4 to P6 of the challenge). Furthermore, the new challenge asks for a solution to add a feature "lastupdate" to objects at any abstraction level. This was rather easily solved since Telos as implemented by ConceptBase assigns a transaction time to any explicit fact stored in the database.

9 Critical reflections

The process modeling challenge is rather demanding for multi-level conceptual modeling because processes are dynamic and their semantics are not as straightforward as the semantics of static data models. The execution of process models leads to artifacts such as instances of tasks. If the process model is about creating software engineering models, then the instances of tasks are associated to models. Thus, the challenge breaks the strict stratification of modeling levels familiar within the UML context. This complication turned out not to be an issue with DeepTelos because the underlying Telos language is level-agnostic. Any notion of level has to be explicitly defined. No axiom in Telos forbids to associate a metaclass to a class, or to an individual object.

Another strength of DeepTelos is the ability to use the rule/constraint/query language to *define the semantics of constructs* and to *analyze multi-level models*. For example, the traceability of model elements discussed in Sect. 6 is defined by simple deductive rules realizing the transitive closure of base relations.

DeepTelos defines rules to derive specializations from powertype-relations ("most-general instances"). This appears elegant at first sight but is also *computationally demanding*. In ConceptBase, a partial evaluation mechanism is employed to compile the rules to a set of simpler rules that are most efficient to evaluate. However, this mechanism reaches its limitation when many rules are generated by the compiler. So, while the requirements of the process modeling challenge were met, the solution with ConceptBase is not satisfactory in terms of efficiency. A large number of rules implies complex computations for large models.

Another suboptimal aspect is that DeepTelos does not drastically reduce *accidental complexity*. In DeepTelos (and other powertype-based

approaches), the numeric levels of classes, attributes and relations are replaced by additional classes in the model (e.g. Task vs. TaskType, Agent vs. AgentType, etc.). In some cases, the relations at the metaclass level needed to be duplicated one level below, including powertype relations between the relations themselves. This should be hidden in more expressive constructs. This paper introduced one such construct, the enumeration class. It allows lifting individual objects to the class level as a generalization of singleton classes. It may indeed be useful to investigate more such constructs. If it is true that level-based approaches better reduce the accidental complexity, then we should investigate whether their constructs can be mapped to powertype-based models. Approaches like DeepTelos that are based on logic are supposedly more elegant in querying large models.

During the development of the solution, some weaknesses of the multi-level partial evaluator of ConceptBase became apparent. While ConceptBase allows changing objects at any abstraction level at any time (including deleting them), this has put the partial evaluator to its limits. It maintains a dependency graph between code that is generated from the formulas to maintain the executable rules. This graph can become cyclic and then becomes prone to let the evaluator run into a loop. This can be avoided by defining the most-general instance relations at the beginning and then not change them afterwards. Still, it was an unpleasant experience. The partial evaluator is also a bit time consuming. It can take a few seconds to compile all modules. Once compiled, the execution is relatively fast on small models.

Different to Datalog, arithmetic functions are supported by ConceptBase. In general, functions can generate new tokens (numbers) and thus compromise the guarantee that Datalog rules are safe, i.e. their extensions can be computed in finite time. In the process modeling challenge, some arithmetic functions had to be introduced, e.g. to compute the actual duration of a task execution. So, in general the DeepTelos solution of the process could be unsafe. A manual inspection is needed to check that the arithmetic functions do not cause any harm.

An interesting exercise was to revisit our old software process data model (see Fig. 3) and reformulate it as a multi-level model. In fact, we did not fully realize back in 1990 that the constructs at the highest abstraction level needed to be instantiated two levels below. The multi-level variant easily solves this shortcoming.

The module facility of ConceptBase was very useful to prepare the solution to this challenge. It allowed separating the definition of DeepTelos from the definitions of the example process modeling languages and their instances. Just by placing a multi-level model under the module for DeepTelos, all its axioms become applicable.

10 Conclusions

We provided a solution for the process modeling challenge. All requirements were met. We also provided an example process execution to highlight how the ACME process can be traced and monitored.

The solution does have some elegance to it, e.g. directly using the terms actor, actor types etc. in the solution. We remain unconvinced that we saved a lot of coding via the DeepTelos multi-level modeling approach. The advantage seems to be more in reuse rather than avoiding accidental complexity. This statement applies to comparison of DeepTelos vs. Telos, not to DeepTelos vs. a set of equivalent two-level models. For example, it was easy to embed an existing BPMN core meta model into the constructs of this challenge by using standard specialization.

A key advantage of ConceptBase was that it naturally supports any number of instantiation levels. Another advantage is that the object "Proposition" is fully accessible to users of ConceptBase to add new abstract constructs extending the attribution, instantiation and specialization constructs.

10.1 Access to the source code

The source code of our solution together with numerous examples and

further documentation is available at http://conceptbase.cc/emisaj2021challenge. See sub-directory SOURCES for the complete source code of all modules listed in Fig. 4. All model diagrams in this paper are created by the ConceptBase graph editor and are also made available as graph files from the web page. Graph files are self-contained views of models stored in ConceptBase.

References

Atkinson C., Gutheil M., Kennel B. (2009) A Flexible Infrastructure for Multilevel Language Engineering. In: IEEE Trans. Software Eng. 35(6), pp. 742–755

Atkinson C., Kühne T. (2001) The Essence of Multilevel Metamodeling. In: UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, pp. 19–33

Atkinson C., Kühne T. (2008) Reducing accidental complexity in domain models. In: Softw. Syst. Model. 7(3), pp. 345–359

Ceri S., Gottlob G., Tanca L. (1989) What you Always Wanted to Know About Datalog (And Never Dared to Ask). In: IEEE Trans. Knowl. Data Eng. 1(1), pp. 146–166

Fonseca C. M., Almeida J. P. A., Guizzardi G., de Carvalho V. A. (2018) Multi-level Conceptual Modeling: From a Formal Theory to a Well-Founded Language. In: 37th International Conf. Conceptual Modeling, ER 2018, Xi'an, China, October 22-25, 2018. Springer, pp. 409–423

Frank U. (2014) Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. In: Business & Information Systems Engineering 6(6), pp. 319–337

Frank U. (2019) The MULTI 2019 Process Challenge - A Solution based on the FMMLx. Presentation at MULTI 2019

Gonzalez-Perez C., Henderson-Sellers B. (2006) A powertype-based metamodelling framework. In: Softw. Syst. Model. 5(1), pp. 72–90

Jarke M. (ed.) Database Application Engineering with DAIDA. Research Reports ESPRIT. Springer

Jarke M., Gallersdörfer R., Jeusfeld M. A., Staudt M., Eherer S. (1995) ConceptBase - A Deductive Object Base for Meta Data Management. In: J. Intell. Inf. Syst. 4(2), pp. 167–192

Jarke M., Jeusfeld M. A., Rose T. (1990) A software process data model for knowledge engineering in information systems. In: Inf. Syst. 15(1), pp. 85–116

Jeusfeld M. A. (2005) Complete List of O-Telos Axioms. Online: http://merkur.informatik.rwth-aachen.de/pub/bscw.cgi/d1228997/O-Telos-Axioms.pdf

Jeusfeld M. A. (2009) Metamodeling and method engineering with ConceptBase. In: Jeusfeld M. A., Jarke M., Mylopoulos J. (eds.) Metamodeling for Method Engineering. MIT Press, pp. 89–168

Jeusfeld M. A. (2011) A Deductive View on Process-Data Diagrams. In: 4th IFIP WG 8.1 Working Conf. on Method Engineering, Paris, France, April 20-22, 2011, pp. 123–137

Jeusfeld M. A. (2019) DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In: 22nd ACM/IEEE MODELS Conf., Companion, Munich, Germany, September 15-20, 2019. IEEE, pp. 66–77

Jeusfeld M. A. (2022) ConceptBase.cc User Manual - Version 8.2. Last Access: http://conceptbase.sourceforge.net/userManual82/CB-Manual.pdf

Jeusfeld M. A., Almeida J. P. A., Carvalho V. A., Fonseca C. M., Neumayr B. (2020) Deductive reconstruction of MLT* for multi-level modeling. In: ACM/IEEE 23rd MODELS Conf., Canada, 18-23 October, 2020, Companion Proceedings, 83:1–83:10

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level Modeling with Most General Instances. In: Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, pp. 198–211

Jeusfeld M. A. et al. (2021) ConceptBase User Manual Version 8.1. Online: http://conceptbase.sourceforge.net/userManual81/

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: Requir. Eng. 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

Lloyd J. W., Topor R. W. (1984) Making Prolog more Expressive. In: J. Log. Program. 1(3), pp. 225–240

Mylopoulos J., Borgida A., Jarke M., Koubarakis M. (1990) Telos: Representing Knowledge About Information Systems. In: ACM Trans. Inf. Syst. 8(4), pp. 325–362

Odell J. J. (1998) Advanced object-oriented analysis and design using UML In: Cambridge University Press chap. Power types, pp. 23–32

Partridge C., de Cesare S., Mitchell A., Odell J. (2018) Formalization of the classification pattern: survey of classification modeling in information systems engineering. In: Softw. Syst. Model. 17(1), pp. 167–203

Rodriguez A., Macias F. (2019) Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: 22nd ACM/IEEE MODELS Conf., Companion, Munich, Germany, September 15-20, 2019. IEEE, pp. 152–163

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: 22nd ACM/IEEE MODELS Conf., Companion, Munich, Germany, September 15-20, 2019. IEEE, pp. 119–127