Multi-level modeling with *LML*

A Contribution to the Multi-Level Process Challenge

Arne Lange*,[a], Colin Atkinson[a]

[a] Institute of Computer Science and Business Informatics, Software Engineering, University of Mannheim, Mannheim, Germany

Abstract. *This paper presents a solution to the MULTI Process Challenge which was first posed to the participants of the MULTI workshop at the MODELS conference in 2019 and subsequently adapted for this special issue of the EMISA Journal. The structure of the paper therefore follows the guidelines laid out in the Challenge description. The models are represented in the Level-agnostic Modeling Language* LML *and the* DOCL *constraint language using the* Melanee *deep modeling tool. After first outlining the case study and documenting which aspects are supported in the* LML *solution, the paper presents multi-level models for both the insurance and the software engineering domains. This is followed by a discussion of the strengths and weaknesses of the approach. The presented model covers all mandatory and optional aspects of the Challenge case study.*

Keywords. multi-level modeling • process challenge • deepOCL

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

1 Introduction

The EMISAJ Multi-level Process Challenge (hereafter referred to as "the Challenge") was initially defined by the organizers of MULTI 2019 as a vehicle for evaluating and comparing multi-level modeling approaches and was subsequently refined for this special edition of the EMISAJ journal by Almeida et al. (2019). This paper presents a solution to the challenge developed using the *LML* deep modeling language supported by the *Melanee* deep modeling tool (Gerbig 2017). This supports a style of multi-level modeling often referred to as "deep (meta) modeling" (Atkinson and Kühne 2001b) since it is based on the use of (a) the deep instantiation mechanism (using potency) to represent ontological classification relationships (Atkinson and Kühne 2001b) and (b) the tenet of strict modeling to define the different levels within a multi-level model (Atkinson and Kühne 2002).

In any modeling approach there are no set-in-stone criteria for judging what constitutes a good model as opposed to a bad model since models can be optimized for different purposes based on multiple criteria (e. g., minimality, readability, maintainability). The deep models presented in this paper, which satisfy all mandatory and optional requirements laid out in the Challenge, have been optimized to showcase *LML* features and thus do not claim to be the best models for a specific real-world modeling use case.

The structure of the paper broadly follows the guidelines set out in the Challenge description. The next section first outlines the technology we use to develop our solutions including the basic principles of our modeling approach, the important concepts in the *LML* and *DOCL* languages and the properties of the supporting *Melanee* tool. Once the technology has been described, Sect. 3 starts the presentation of our solutions by describing the level-spanning elements of the models, including the level-spanning linguistic metamodel and the basic rules by which elements at one level can be

* Corresponding author.
E-mail. lange@informatik.uni-mannheim.de

related to the elements at the level above. Sect. 4 then presents the top level ontological level in our solution(s) which describes the general purpose process modeling language which is independent of any particular domain or example. The use of this language in the two applications domains is described in the following two sections. Sect. 5 deals with our solution for the insurance domain and Sect. 6 for the software engineering domain. Sect. 7 continues by addressing the mandatory and optional discussion points outlined in the Challenge description before Sect. 8 continues with a deeper discussion of one of the main weaknesses of our technology and a presentation of some potential solutions. Finally, Sect. 10 concludes with some closing remarks.

2 Technology - Applied Multi-Level Modeling Approach

In this section, we characterize the multi-level modeling approach we used to model our solutions.

2.1 Deep Modeling

Since the basic tenets of Multi-level Modeling (*MLM*) approaches were first identified around two decades ago, a large number of different *MLM* languages have been proposed. The common goal of these languages is to support the representation of domain information using multiple classification levels rather than just the two levels simultaneously supported in most traditional modeling environments. The variant of multi-level modeling used in this paper is sometimes known as "deep modeling" or "deep meta-modeling" (De Lara et al. 2014a), because one of its four core principles is referred to as "deep instantiation" (Atkinson and Kühne 2001b). The basic ingredients of deep modeling are:

1. Orthogonal Classification Architecture (*OCA*)
2. Strict modeling
3. Clabjects
4. Deep instantiation

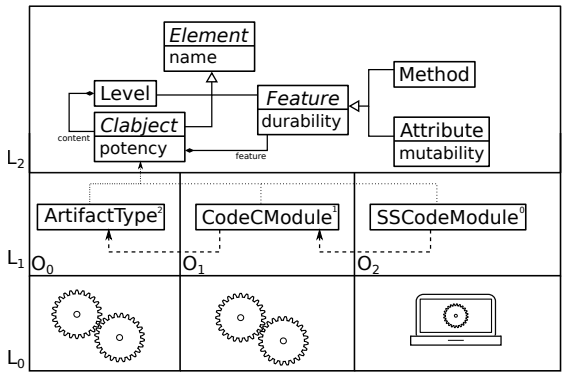


Figure 1: Orthogonal Classification Architecture

Orthogonal Classification Architecture

Perhaps the most fundamental principle used in deep modeling is that all modeling content exists within two orthogonal classification dimensions – a linguistic dimension which describes *how* domain concepts are represented, linguistically, and the ontological dimension which captures *what* properties and relationships those domain concepts have. This means that the domain content within a deep model can have two types, an ontological type and a linguistic type, as shown schematically in Fig. 1.

The model at the L_2 level is known as the linguistic model and describes the abstract and concrete syntax of the language used to represent all domain content across all ontological levels in L_1. In our case, the language is *LML*. The L_2 level of Fig. 1 contains a highly simplified depiction of the core linguistic concepts of *LML*, represented using a UML-like language.

Strict Modeling

The second fundamental ingredient of deep modeling is that the "levels" in the ontological dimensions are strictly and exclusively organized by the classification (i. e., *instanceOf*) relationship and that the instances of a model element are always regarded as occupying the level below that model element's level. This can be seen in L_1 of Fig. 1 which depicts three ontological levels, each containing one clabject, where *SSCodeModule* is an *instanceOf* CodeModule, which in turn is an

instanceOf `ArtifactType`. The *instanceOf* relationship is the basis for establishing the ontological levels in the strict modeling approach. A second important principle of strict modeling, therefore, is that *instanceOf* relationships are the only kind of relationship that can cross level boundaries.

Different numbering conventions can be chosen to label the ontological levels, depending on whether the emphasis is on defining a generic framework intended for reuse in different domains, with different numbers of derived classification levels, or whether the emphasis is on defining a deep model for a fixed domain where the number of levels is unlikely to change. In the former, it is more convenient to number the ontological levels from the most abstract to the most concrete (i. e., O_0 is the top level) while in the latter it is more convenient to number them from the most concrete to the most abstract (i. e., O_0 is the most concrete). We use the former in our solution.

Note that strict modeling does not require every model element to have an ontological type, so it is not necessary for a model element to have a type at the level above. Strict modeling merely requires that if a model element has an ontological type, that ontological type must be at the level above.

Clabjects

The third fundamental ingredient of deep modeling is that, in general, the elements in a deep model are both types and instances at the same time, as can be seen in Fig. 1. The model element, CodeModule, in the middle level is an instance of the model element, `ArtifactType`, at the level above as well as the type of the model element, *SSCodeModule*, at the level below. Since UML typically refers to types as "classes" and instances as "objects", to capture this unification of the two concepts within a single abstraction, model elements in deep models are often called "clabjects" (a conflation of Class and Object). Note that since clabjects do not have to have ontological types, and types do not have to have any currently existing instances, clabjects that represent just types or just instances can exist at any level.

Deep Instantiation

Combining the notions of types and instances (i. e., classes and objects) into the unified notion of clabjects has the advantage that model elements can play both roles, thereby reducing clutter and unnecessary complexity in models. However, without additional mechanisms to characterize the resulting type and instance "facets" of clabjects it would not be clear to what extent, and how, individual clabjects can play type and/or instance roles. The third fundamental ingredient of deep modeling languages is therefore a mechanism to represent the "vitality" of clabjects – namely, to what extent, and how, the type facet of clabjects influence their direct offspring (i. e., their direct instances, over multiple classification levels).

Potency: The oldest and most fundamental vitality property in *MLM* is the so-called "potency" property of a clabject which is a measure of the number of classification levels over which a clabject can have direct instances. It is captured as an integer-valued linguistic attribute (or trait) of a clabject that adheres to two fundamental rules:

1. the potency of a clabject cannot be less than 0,
2. the potency of a direct instance of a clabject must be less than the potency of that clabject.

The notion of potency was first introduced as part of the so-called "deep instantiation" mechanism (Atkinson and Kühne 2002) which distinguishes deep modeling from traditional "shallow" modeling approaches and ultimately gives the former its name. In its initial form, the deep instantiation mechanism was based on the principle that the potency of an instance of a clabject must always be exactly one less than the potency of that clabject. However, a more relaxed interpretation of potency (Kühne 2018) requires only that the potency of an instance of a clabject be less than the potency of that clabject (subject to the two constraints defined above).

The potency trait captures the most fundamental type characteristics of clabjects – namely, how many lower levels the clabject can influence by transitive instantiation chains. However, it does

not characterize the nature of that influence in terms of the attributes that those instances can have and the values those attributes can take. The *LML* version of deep modeling used in this paper addresses this question by introducing two further vitality properties, called "durability" and "mutability", which govern the existence of ontological attributes of clabjects and the values of those attributes, respectively. In previous publications, these were often referred to as "forms of potency". However, to avoid confusion with the true potency property describe above, we now prefer to refer to these three linguistic attributes as vitality properties (Lange and Atkinson 2019).

Durability: Durability is a linguistic property of the ontological attributes of clabjects which characterizes their endurance over instantiation steps. Like potency, it is represented by a non-negative integer, but unlike potency, an instance of an attribute must have a durability that is exactly one less than the durability of that attribute. Thus, an instance of a clabject that has an attribute of durability zero need not have an instance of that attribute.

Mutability: Mutability is a linguistic property of the ontological attributes of clabjects that characterizes how their values can be changed over instantiation steps. Like potency and durability, mutability is a non-negative integer. Moreover, with one exception, the rules for mutability over instantiation steps are the same as those for durability – the mutability of an instance of an attribute must be one lower than the mutability of that attribute. The exception occurs when the mutability of the attribute is already zero, in which case the mutability of instances of the attribute must also be zero. Zero is a key mutability value of an attribute, therefore, it means that the instances of that attribute must have the same value.

2.2 Level-Agnostic Modeling Language

The concrete deep modeling languages we use to describe our solution are the Level-Agnostic Modeling Language (*LML*), and deep *OCL* (*DOCL*), both of which are based on *OCA*. *LML* was developed to support the description of domain model content within L_1 in a way which is:

1. UML-like,
2. level-Agnostic,
3. minimalistic.

The underlying goal of *LML*'s general purpose syntax is to support the creation of deep models which have the basic look-and-feel of UML models, and thus are intuitive to mainstream modelers and software engineers. However, *LML* was designed to achieve this goal in a way that is level-agnostic and minimalistic. The language is level-agnostic in the sense that all constructs defined in the linguistic model, which are therefore available at all ontological levels, are applied and represented in the same way at all levels. For example, in contrast to UML, there is no difference in the concrete syntax used to represent classes and objects, which means that clabjects with all possible combinations of types/instance characteristics can be represented using the same notation.

2.3 Deep OCL

The deep *OCL* (*DOCL*) dialect integrated into *Melanee* extends the *OCL* 2.4 (Object Management Group 2021) language specification with additional features to write constraints in the context of deep models. Most importantly it is "level-aware", which means that *DOCL* expressions can explicitly refer to, and navigate over, the ontological as well as the linguistic dimensions. *DOCL* also adds additional features such as reflective queries for specific types over both dimensions. Normal *OCL* typing queries can still be used wherever desired, but additional "level-aware typing queries", such as *isDeepInstaceOf()*, *isDeepTypeOf()* or *isDeepKindOf()*, allow typing queries to transcend multiple classification levels. In comparison to standard *OCL*, where statements are purely descriptive and have no effect on the model, *DOCL* is also able to manipulate models. This can be done via a *derive* or *init* constraint, where the value of an attribute is changed depending on the

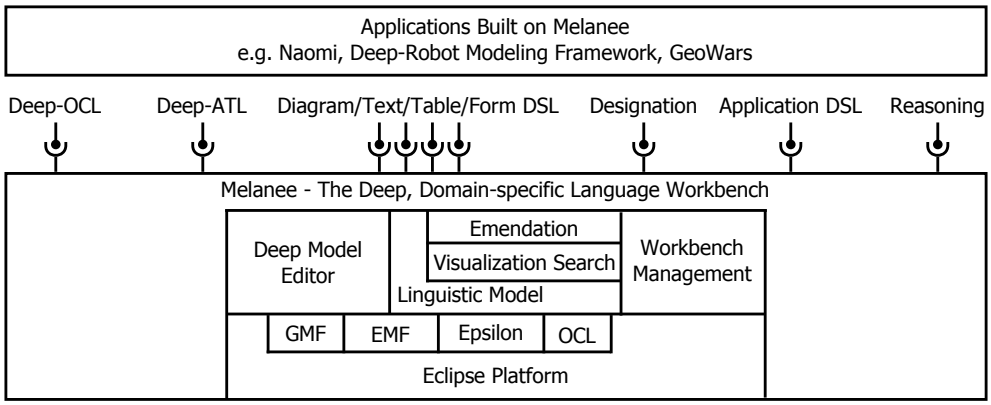


Figure 2: Melanee Architecture Overview (Gerbig 2017)

state of the model. In fact, this technique is used to fulfill one of the requirements of the Challenge, which will be part of the solution description. In a deep environment the *invariant* constraints can be evaluated at modeling time, instead of just at run time as in a traditional *OCL* environment.

In order to switch to the linguistic dimension the '#' symbol is used. A subsequent '#' symbol sets the context back to the "normal" ontological context. Whenever the linguistic dimension is queried, attributes and methods from this dimension (i. e., the linguistic model) can be accessed.

All *DOCL* constraints have a level "modifier" that determines the levels at which the constraint is executed. For example, a constraint defined at level O_0 can apply to (i. e., be executed on) level O_1 or level O_2 or both. It is also possible to define an interval that includes all levels below the level the constraint is specified at. This is indicated with brackets behind the context specification of the constraint. For example, if the execution interval should involve all levels, the execution specification has the form '(0,_)'

2.4 Melanee

The concrete tool we used to create our *LML* models is the *Melanee* tool developed at the University of Mannheim (Gerbig 2017). The tool is based on the ECLIPSE platform (Eclipse Foundation 2021) and, as illustrated in Fig. 2, uses that platform's plugin features to provide a full, multi-level modeling experience that includes access to *DOCL* and deep ATL (a deep variant of the well-known ATL transformation language).

As well as supporting the general-purpose *LML* modeling notation used in this paper, *Melanee* also allows deep models to be visualized in various other forms including textual, form-based, tabular and domain-specific notations. *Melanee* also provides numerous reasoning and checking services for deep models, including the so-called "emendation" service which checks that clabject vitality properties are consistent and offers support for making corrections where necessary. Finally, although *Melanee* provides access to the aforementioned features optimized for deep modeling, since the linguistic model is essentially realized as an ECore model (i. e., as an instance of the ECore metamodel underpinning the Eclipse Modeling Framework (EMF)), the rich set of modeling languages and tools offered in EMF can be used on *Melanee* models, albeit in a level-unaware way.

3 Level-Spanning Artifacts

3.1 Linguistic Model

This section describes all level spanning concepts underpinning our *LML*-based solution to the Challenge. This consists of two parts, the linguistic model itself, taking the form of a class diagram, and a set of constraints, taking the form of *DOCL* expressions. Each part is described in a dedicated section below.

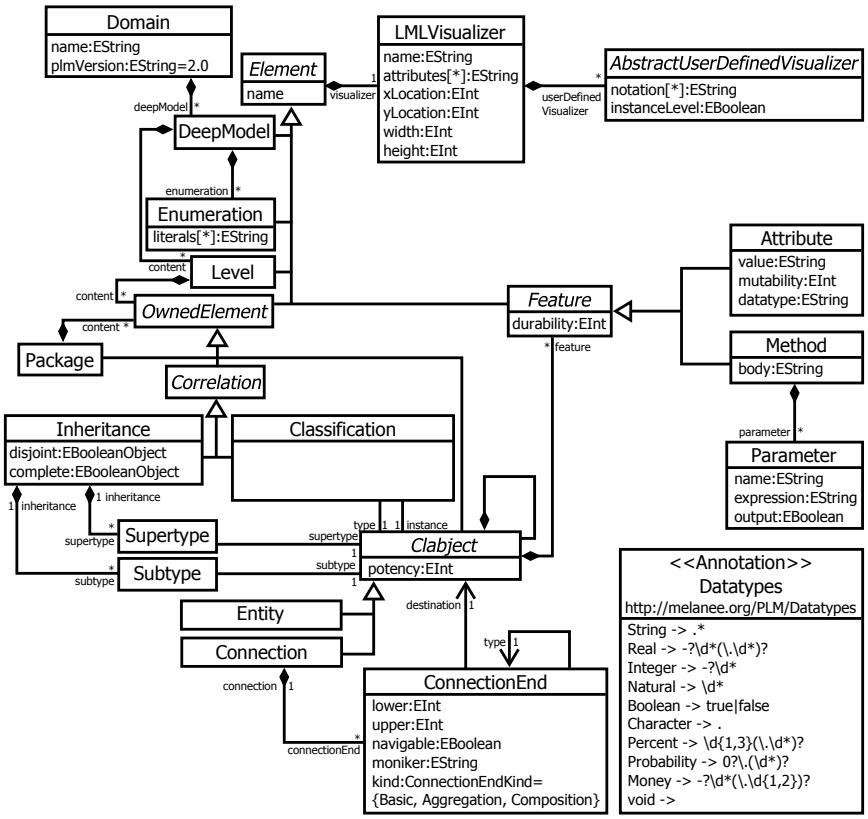


Figure 3: Pan-level model for LML that is used in Melanee

The linguistic model for *LML* deep models, describing the abstract syntax for the concepts explained in the previous section, is depicted in Fig. 3 in the form of a UML class diagram. This shows that all the model elements comprising an *LML* deep model for a particular *Domain* are direct or indirect instances of the abstract class *Element*. A deep model, which is an instance of the class *DeepModel*, can contain any number of *Level* instances and a *Level* can accommodate an arbitrary number of *OwnedElement* instances, which is the superclass of *Clabject*, *Classification*, *Inheritance* and *Feature*. At the heart of the model is the abstract class, *Clabject*, which has two subclasses, *Connection* and *Entity*. *Clabject* instances can have zero or more *Feature* instances which can either be *Attributes* and *Methods*. Note that clabjects can also contain other clabjects. This is useful, for example, in component diagrams where a clabject representing an outer component can contain clabjects representing inner components, or in activity diagrams where a clabject representing a swimlane can contain clabjects representing activities.

Two kinds of set-theoretic relationships, known as *Correlations* can be represented between clabjects, *Inheritance* relationships and *Classification* relationships. *Inheritance* relationships, which can only exists between clabjects at the same level, relate clabjects playing *Superytpe* or *Subtype* roles. *Classification* relationships, in contrast, can only exist between clabjects in adjacent levels, and indicate that one clabject (the one at the lower level) is an instance of another clabject (the one at the higher level).

3.2 Level-Spanning Constraints

In this subsection we present two level spanning constraints that impose additional rules on the way the content within one ontological level is derived

from the content in the level above. These rules go beyond the basic rules of deep modeling, and ensure that the concepts defined in one level are used in a particular way in the level below.

3.2.1 Isonymic Instance Rule

The first rule ensures that all ontological instances of a clabject are "isonyms" of that clabject. This means that they must have all the features required by the clabject, conformant with the durability values of its features, but no more features (i. e., attributes and methods) (Atkinson et al. 2011). While this constraint is an inherent "paradigm" of traditional constructive modeling, in deep modeling it is not automatically present because clabjects can have more features than required by their ontological type by virtue of the fact that they also have a linguistic type. Therefore, if this paradigm is applied in a particular modeling context, such as a solution to the modeling Challenge, it needs to be explicitly declared, even if only informally. *DOCL*'s ability to express level spanning constraints allows such fundamental level usage constraints (i. e., paradigms) to be defined formally.

The notion of isonymic and hyponymic instances of clabjects was introduced to distinguish between instances that only have the features required by the clabject, and no more (i. e., isonyms) from those that have more features than are strictly needed to satisfy the intension of the clabject (i. e., hyponyms) (Atkinson et al. 2011). The following two constraints, therefore, enforce our so-called "isonymic instantiation" rule which requires that all clabjects within a deep model that have an ontological type should be isonyms of that type.

```
context DeepModel(0,_)
inv PAN-1: Clabject -> forAll(select(c|c.#
    getFeature()# -> select(f|f.#
    getDurability()#
> 0)) -> size() = c.#getDirectInstances()#
    -> select(cc|cc.#getFeature()#) -> size
    ())
```

The first constraint, called 'PAN-1', compares the size of the attribute collection of the type clabject and all the direct instances of that clabject. First, the reflective query obtains the collection of all clabjects in a deep model and, for each clabject in the collection, the number of the contained attributes and methods that have a durability value greater than zero, including all inherited features, and states that all direct instances must have the same number of features.

```
context DeepModel(0,_)
inv PAN-2: Clabject -> forAll(select(c|c.#
    getFeature()# -> select(f|f.#
    getDurability()# > 0) ->
    collect(#name#)) -> includesAll(c.#
    getDirectInstances()# -> select(cc|cc.#
    getFeature()#) -> collect(#name#)))
```

The second constraint called 'PAN-2' is very similar to the 'PAN-1' constraint, but instead of comparing the size of the collection of features, this constraint compares the content of the collections. Every direct instance of a clabject must not introduce additional features that are not present (with a durability value greater than zero) in the direct type. Instead of directly comparing features, we just compare their linguistic attribute names.

Note that it is also possible, if desired, for a modeler to apply this isonymic instance requirement more selectively by defining similar constraints at the scope of individual clabjects rather than for all clabjects.

3.2.2 No Ontologically-Untyped Connections Rule

The previous two constraints ensure that if a clabject has an ontological type, it must conform to that type as an isonymic instance. However, they do not rule out the possibility that clabjects that do not have an ontological type can be added to a lower level. This is possible in deep modeling because the fundamental form of model elements is defined by their linguistic type rather than their ontological type, meaning that the latter is strictly speaking, optional. However, in many deep modeling scenarios, for example, when an ontological level is meant to define a strictly applied domain-specific language, it is desirable to adopt a rule

(i. e., paradigm) in which every clabject has to have an ontological type at the level above.

Such a rule could easily be defined in a particular modeling context to ensure that all clabjects, except those at the top level, have ontological types. However, for our purposes, this rule would be too strict since we need to provide modelers with the flexibility to introduce some unforeseen types in a level. In the context of our solution to the Challenge, the balance between control/flexibility required in the challenge description is best achieved by a slightly more relaxed modeling rule – namely that all connections in a deep model (except those at the top ontological level) must have an ontological type. This ensures that applications of the top level, which defines the basic process modeling concepts required in the challenge, adhere to all domain knowledge/rules captured by the connections (e. g., the well-formedness of task composition), but provides the freedom for modelers to introduce new (ontologically untyped) entity clabjects to reflect the idiosyncrasies of the scenario of interest.

The following constraint, therefore, enforces the so-called "no ontologically-untyped connections" rule which requires that all connections, except those at the top level, have an ontological type. In effect, this means that no new connection kinds can be added to the deep model beyond those in the top level. The constraint begins by collecting all *Connections* present in the deep model, and then rejecting (i. e., removing) those that reside at the most abstract level. For all the remaining connections, the constraint requires that the direct type of each connection exists. The function calls that start and end with '#' retrieve the necessary information from the linguistic dimension. The query for all direct types of a connection returns a set of connections because the scope of this function is to retrieve a chain of types up to the most abstract level. It is important, therefore, that the collection of types must not be empty.

```
context DeepModel
inv PAN-3: Connection -> reject(c|c.#
    getLevel()#.name = 'O0') -> forAll(c|c.#
    getDirectTypes()# -> size() > 0)
```

4 General Process Modeling Language

The description of the Challenge presented in Almeida et al. (2019) encompasses three different characterization contexts, some explicitly and some implicitly. Our solutions for the two scenarios referred to in the Challenge are both comprised of three ontological levels, with the top level being shared. This level addresses the most general and most extensively described characterization context - that of characterizing (i. e., defining the concepts appearing in) domain-independent process definitions (Sect. 2.2 of Almeida et al. (2019)). This section presents the common, top (O_0 level) model, and associated constraints, which capture the generic set of process description features called for in the Challenge description. The *LML* model is described in the following subsection, while the constraints are presented in the subsequent section that discusses how each requirement is fulfilled.

4.1 Generic Process Metamodel

The *LML* ontological metamodel, at the O_0 level, supporting the aforementioned requirements in our solution is depicted in Fig. 4. The most abstract concept in this model is the *Element* clabject which is the superclass of all clabjects in the model. Since it can have no direct instances of its own, it is an abstract class. The clabject that represents the container for all elements used to describe processes, `ProcessType`, is also a subclass of `Element` in an application of the composite design pattern (Gamma et al. 1994). The potency value of `ProcessType` is '2' so that it can be instantiated over two consecutive lower levels. Actor represents the human-played role responsible for defining instances of a particular kind of `Element`, `TaskType`, at the O_1 level below. Since instances of Actor are therefore only needed at the O_1 level, the potency of Actor is '1'. The Actor clabject is able to create

tasks through the *createdBy* relationship on the level directly below.

There are five basic kinds of elements that can be contained in a process type, which are modeled as subclasses of `Element` – `ControlEventType`, `ActorType`, `TaskType`, `ArtefactType` and `ArtefactKindType`. The first three are abstract classes and therefore have a potency of '0' while the last two are concrete classes with a potency of '2'. `ControlEventType` has four subclassses, two of which are concrete. `StartEventType` and `FinalEventType` have a potency value of '2' and the other two are abstract superclasses, i. e., `SplitEventType` and `JoinEventType` with a potency of '0'. The four concrete subclasses of the split and join events are `AndJoin` and `AndSplit`, `OrJoin` and `OrSplit`. The `StartEventType` clabject has to be present in every instance of a process type exactly once and the `FinalEventType` has to be present at least once. The split and join events can participate in the *followedBy* relationship that is defined on `ProcessElement` in combination with any `TaskType`.

`ActorTypes` is specialized by two classes, i. e., `JuniorActorType` and `SeniorActorType`, while `TaskType` has three subclasses – `NormalTaskType`, `CriticalTaskType` and `ValidationTaskType`. All of the specialized clabjects have a potency value of '2'. `TaskType` itself is impotent and contains three attributes which are *expectedDuration* of type integer, `beginDate` of type string and `endDate` of type string. The durability value of all three attributes is '2', e. g., they are present in all instances of subclasses of this clabject. They also have mutability values of '2' with the exception of *expectedDuration*. This attribute has a mutability value of '1' which means its value can be changed at the level immediately below but not at the levels below that. Every task can produce artifacts that can be of any kind. This is represented by the clabjects `ArtifactType` and `ArtifactKindType` which are connected to `TaskType` by the *producedBy* and *usedBy* connections. It is the *kind* relationship that connects the artifact to the specification of what kind of artifact it is.

4.2 Fulfillment of the Requirements

This subsection describes how each of the requirements, defined in the Challenge, are satisfied by the O_0 level of our solution, with suitable *DOCL* constraints being introduced where necessary.

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:

This is supported by the composition relationship between `Element` and `ProcessType`. Every instance of `Element` is contained in one instance of `ProcessType`, and every instance of `ProcessType` contains at least three `Elements`. The following constraint ensures that at least one of these contained elements is a `TaskType`.

```
context ProcessType(1,2)
inv: self.content -> exists(element|
    element.deepOCLTypeOf(TaskType))
```

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*:

The sequencing relationships between task types is achieved by means of the *followedBy* relationship between `ProcessElements` while the split and join gateways are realized by dedicated clabjects which are subclasses of `ControlEventType`. Instances of *TaskType* must be involved in two *followedBy* connections. One connection, in which it participates as the target, is to the `ProcessElement` that precedes it, and the other, in which it participates as the source, is to the `ProcessElement` that follows it.

```
context TaskType(1,2)
inv taskFollowedBy: self.source -> size()
    = 1 and self.target -> size() = 1
```

The instances of `StartEventType` are not allowed to participate in a *followedBy* relationship as the target, and must therefore be the beginning of a process. It must reach exactly one

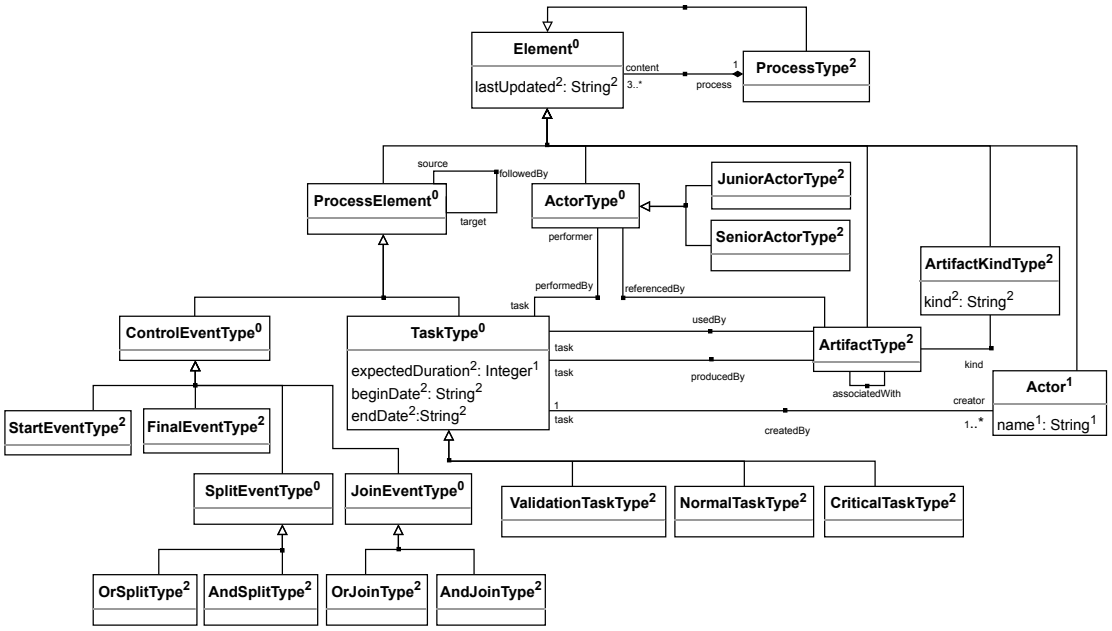


Figure 4: Level O_0

element through the *followedBy* connection, where it is the source of the connection.

```
context StartEventType(1,2)
inv start: self.source -> size() = 0 and
    self.target -> size() = 1
```

The instances of `FinalEventType` are not allowed to have a follower, and must therefore be the end of a process.

```
context EndEventType(1,2)
inv end: self.source -> size() = 1 and
    self.target -> size() = 0
```

No instance of `SplitEventType` is allowed to have only one incoming connection but have to have at least two outgoing connections.

```
context SplitEventType(1,2)
inv split: self.source -> size() = 1 and
    self.target -> size() => 2
```

For the instances of `JoinEventType` the rule on how many outgoing and incoming connections they can have is exactly the reverse of split events.

```
context JoinEventType(1,2)
inv join: self.source -> size() => 2 and
    self.target -> size() = 1
```

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):

To simplify the definition of these well-formedness rules, we avoid the introduction of dedicated clabjects for initial task types and final task types, since these would also need to be distinguished as being normal, critical and validation tasks, and instead we identify initial and final tasks by their connection to start and finish control event types respectively (i. e., `StartEventType` and `FinalEventType`). A task type is therefore an initial task type if it is the target in a *followedBy* connection with an instance of `StartEventType`, and is a final task type if it is the source in a *followedBy* connection with an instance of `FinalEventType`. The following constraint ensures that a `ProcessType` has the correct number of `StartEventTypes` and `FinalEventTypes`.

```
context ProcessType(1,2)
inv: self.content -> one(element|element.
    deepOCLTypeOf(StartEventType)) and
    self.content -> exists(element|element
    .deepOCLTypeOf(FinalEventType))
```

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*:

This requirement is supported by the mandatory *createdBy* relationship between `TaskType` and Actor which has a multiplicity constraint with a lower bound of '1' at the Actor end.

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*. For example, in the *XSure* insurance company, only a *claim handling manager* or a *financial officer* may *authorize payments*:

This feature is enabled by the *performedBy* relationship between `TaskType` and `ActorType` which has 0..* multiplicity and thus is optional. The specific authorizations applicable in a particular scenario are established by the instances of these clabjects.

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*):

Our approach supports this capability by allowing constraints to be defined at the O_1 level that control which instances of specific `ActorTypes` can enter into *performedBy* relationships with specific `TaskTypes` at the O_2 level. Such constraints therefore effectively declare which individuals (identified by their names) are authorized to perform which tasks. The constraint used to meet requirement S7 is an example.

P7) For each *task type* (such as *authorize payment*) one may stipulate the *artifact types* which are *used* and *produced*. For example, *assess claim* uses a *claim* and produces a *claim payment decision*:

This requirement is supported by the *usedBy* and *producedBy* relationships between `TaskType` and `ArtifactType`.

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

This is modeled by the *expectedDuration* feature which all offspring of `TaskType` receive by virtue of the fact that it has durability '2'. The mutability of the *expectedDuration* is set to '1' so that specific instances of `TaskType` for a particular scenario at the O_1 level, can change it to the appropriate value for that `TaskType`. However, because the mutability of the *expectedDuration* attributes at O_1 then become '0', instances of a specific `TaskType` at the O_2 level cannot assign a new value to *expectedDuration*, they must retain the value set at O_1. The *expectedDuration* of `TaskType` at the O_0 level is set to undefined.

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*:

The concept of critical task types and senior actor types are modeled by the `CriticalTaskType` and `SeniorActorType` subclabjects of `TaskType` and `ActorType`, respectively. The fact that critical task types can only be performed by senior actor types is captured by the following constraint on the *performedBy* relationship between task types and actor types.

```
context CriticalTaskType(1,2)
inv: self.performer -> forAll(p|p.
    deepOCLKindOf(SeniorActorType)) and
    self.target.isDeepOCLTypeOf(
    ValidationTaskType)
```

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

This is supported by the basic mechanics of the deep modeling approach which allows the instances of the O_0 level metamodel, at O_1, to be instantiated again at O_2. The enactment of a process type is captured by the enactment of the specific task types it contains, which in turn is captured by their instantiation at the O_2 level.

P11) Each process comprises one or more tasks:

The containment relationship multiplicities define that an instance of `ProcessType` must contain at least one indirect instance of `Element`. But in order to ensure that at least one indirect instance of `TaskType` is present in the containment the following *DOCL* constraint is needed.

```
context ProcessType(1,2)
inv: self.content -> select(c|c.
    isDeepKindOf(TaskType)) -> size() > 0
```

P12) Each *task* has a *begin date* and an *end date*. (e. g., *Assessing Claim 123* has *begin date 01-Jan-19* and *end date 02-Jan-19*):

This requirement is realized by means of the *beginDate* and *endDate* attributes of `TaskType` which capture the start and end time of `TaskType` instances and their instances, in turn. The durability and mutability values of these attributes are set to '2' so that their values can be changed at any ontological level.

P13) *Tasks* are associated with *artifacts* used and produced, along with performing *actors*:

This requirement is supported by the *usedBy* and *producedBy* connections between `TaskType` and `ArtifactType` as well as the *producedBy* connection between `TaskType` and `ActorType`

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

The ability to stipulate artifact types used and produced by a specific task type is supported by the *usedBy* and *peformedBy* relationships mentioned above. The actual stipulation for a particular scenario takes place at the O_1 level.

P15) An actor may have more than one actor type (e. g., Senior Manager and Project Leader.):

The ability to declare that there are specific actor types that can perform multiple roles is supported by the multiple inheritance capability at the O_1 level.

P16) Likewise, an artifact may have more than one artifact type:

The ability to declare that a specific artifact can be regarded as being instances of multiple artifact types is supported by the multiple inheritance capability at the O_1 level.

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:

Our strategy for supporting authorization is described in the discussion for P6 above. This requirement does not stipulate at which point (i. e., in which characterization context) the authorization takes place. We have therefore adopted the approach that the authorization is declared at process enactment time (i. e., in the process enactment characterization content) and therefore uses an actor's *instanceOf* relationship to an actor type to designate authorization to perform instances of the task type performed by that actor type.

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.

This is naturally supported by the use of the specialization relationship at the O_1 level.

P19) All modeling elements, at all levels, must have a last updated value of type timestamp.

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:

The existence of this value for modeling elements is captured by the *lastUpdated* String attribute of the *Element* class at the root of the inheritance hierarchy, which has durability and mutability of '2'. This, in turn, means every model element in the deep model has this attribute and can set it to any value. The actual mechanism for arranging for this attribute to obtain the correct value, automatically, when an update event occurs is the *oclGetCurrentDate* operation of *DOCL* which returns the current date as a *String*. *Melanee* can be configured to trigger such an update whenever a model element is edited which automatically sets the correct timestamp as the value of this attribute, but this is beyond the scope of the paper.

4.3 Domain-specific Notation

Although the challenge description does not explicitly require support for the ability to visualize models using domain specific concrete syntax, this capability has a significant effect on the understandability of a model, especially in the process modeling domain where the ordering constraints between task types need to be visualized as clearly as possible. The value of domain specific notations for achieving this is conveyed explicitly in the Challenge description by its use of an intuitive, domain-specific process modeling notation to represent the basic form of the example software engineering process – the ACME software engineering process, as shown below.

The *OCA* modeling architecture that underpins the deep modeling approach used in our solutions provides a highly flexible and extensible basis for supporting the definition and use of domain specific notations (i. e., concrete syntaxes), and their co-use with the underlying general purpose syntax. This flexibility stems from the fact that model elements, in general, have two classifiers, an ontological one and a linguistic one, both of

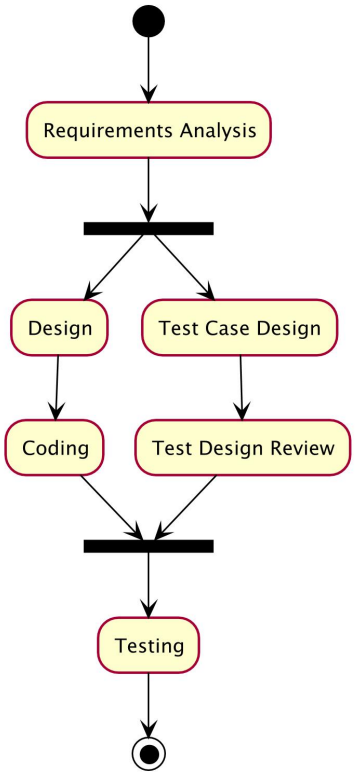


Figure 5: Domain specific notation shown in the process challenge description

which can be used to associate concrete symbols with the model element.

Fig. 6 shows, schematically, how the *Melanee* tool used to define our solutions allows elements in the O_0 process metamodel in Fig. 4 to be augmented with domain-specific concrete syntax (i. e., symbol definitions) facilitating the domain specific visualization of their instance at O_1. The details of how concrete syntax symbols are defined and associated with clabjects and their elements is beyond the scope of this paper[1] , but the basic idea is that a modeler can supply one or more alternative symbols by which clabjects and their deep instances can be visualized. By making the domain symbol assignments depicted in Fig. 6 by means of the cloud symbol, instances of the generic process metamodel shown in Fig. 4 can be

[1] Further details are available in Gerbig (2017)

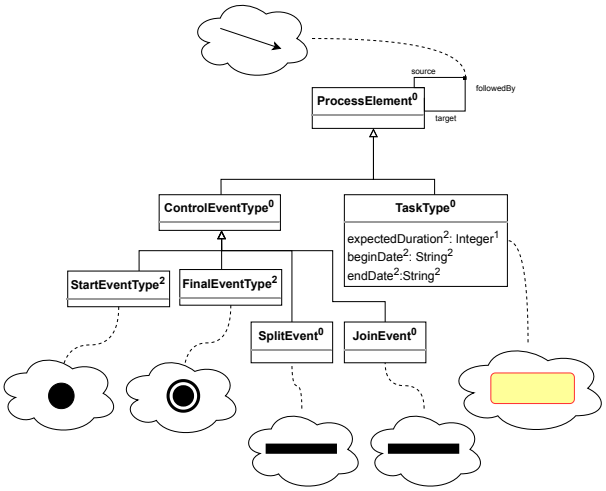


Figure 6: Visualizers for the domain specific notation

visualized in the notation used in the Challenge description, reproduced in Fig. 5.

5 Archisurance Application

The Challenge invites authors to apply the general process modeling language defined in the previous section in two alternative scenarios, one in the software engineering domain and the other in the insurance domain, with the focus on the former. Although it is not mandatory, in this section, we include an example from the insurance domain to show how the aforementioned general process modeling language can be instantiated and used in multiple domains. The main goal of the insurance example is to illustrate the *Melanee* tool's flexible support for the definition and use of domain-specific concrete syntax.

In order to make the example as realistic as possible, we show how a process from the Archisurance example (Jonkers et al. 2012), which is used to illustrate the use of the ArchiMate language (Lankhorst et al. 2009) for Enterprise Architecure Modeling (EAM), can be modeled using our approach. Although fictitious, care has been taken to make the Archisurance example as realistic as possible so that it can be used in accredited ArchiMate training courses in the context of the TOGAF framework (The Open Group 2021). The process that we focus on in this paper is the handle claim process whose original depiction in the ArchiMate language is reproduced in Fig. 7.

5.1 Handle Claim Process Description

The Archisurance Handle Claim process describes how the company processes insurance claims made by policy holders. When a claim is received, the first task to be performed is the Capture Information task in which all relevant information about the triggering incident is obtained. This is followed by the Notify Additional Stakeholders task in which all parties involved in the incident, as well as the owners of the artifacts involved, are informed. The next task is the Validate task which checks whether the claim is compatible with (i. e., supported by) the policy against which the claim is being made. This is then followed by the Investigate task in which the veracity of the events and damage described in the claim are confirmed. Finally, once all these steps have been completed a decision is made about whether to accept or reject the claims. If the claim is accepted, the corresponding payment is made to the policy holder, and if not, the claim is rejected.

The ArchiMate description of the Archisurance Handle claim process shown in Fig. 7 follows many of the same basic principles described in Sect. 2.2 of the process challenge and embodied in the model described in the previous section. The flow between the task types, which are the focus of the model, are described using sequencing and or-split ordering constraints, and events are used to start and finish the process. Finally, all the process model elements are contained within a container bearing the name of the process. Even the concrete syntax is quite similar to the domain-specific notation utilized in the Challenge description. Different symbols are used for start and finish events and or-split events, and the symbol for task types has a small arrow icon in the corner, but other than that the notations are quite similar.

Nevertheless, the ArchiMate model in Fig. 7 does not adhere to all the rules defined in the general process modeling language described in the previous section. More specifically:

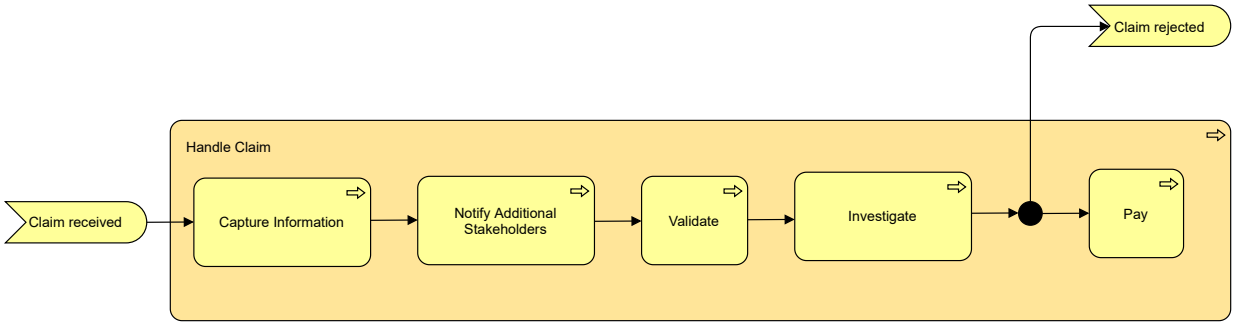


Figure 7: ArchiMate Handle Claim Process

- the start event Claim received and final event Claim rejected are not contained by the task type, Handle Claim,
- there is no task type between the or-split event and the Claim rejected event,
- the Pay task type is not followed by another process element as it should be because it is not a final event,
- none of the task types have associated actor types,
- Investigation is a critical task type but is not followed by a validation task.

Fig. 8 illustrates a slightly modified representation of the Archisurance Handle Claim process type which is modeled by, and thus conforms to, our general process modeling language. The model, therefore, represents an instance of our O_0 ontological model at the O_1 level. The model is depicted using a mixture of *LML* general purpose notation and the domain specific notation defined in Sect. 4.3, and contains the minimum number of changes requires to make it conform to the general process ontological metamodel:

- the start event Claim received is now contained by Handle Claim and is represented using our domain-specific notation for start events,
- the final event Claim rejected is now contained by Handle Claim and is represented using our domain-specific notation for final events,
- the or-split event is now represented using our domain specific notation,
- the task type Reject Claim has now been added between the or-split event and final event Claim rejected,
- the FinancialOfficer actor has been added to perform the Pay task,
- InvestigationReport has been added as the artifact type *producedBy* the *Investigate* task type,
- WrittenReport has been added as the the artifact kind type of the InvestigationReport artifact type,
- the task type InvestigationValidation has been added after the Investigate task type to satisfy the corresponding rule,
- the *TaskDesigner* Actor has been added as the designer of all of the task types defined in the model.

5.2 Domain Specific Notation

In Fig. 8, our version of the *Handle Claim* process is represented using a mix of the standard *LML* concrete syntax and the domain specific-notation defined in Fig. 6. The optimal mix depends on the goals and skills of the stakeholders who need to work with the model, and can range from exclusively *LML* concrete syntax to the maximum use of domain-specific notation (e. g., by representing the task types using the available domain-specific symbols as well). The advantage of handling concrete syntax in a "deep" way, as well as the abstract syntax, is that any part of a model (ranging from a single model element to the whole model) can be "toggled" between the general purpose and

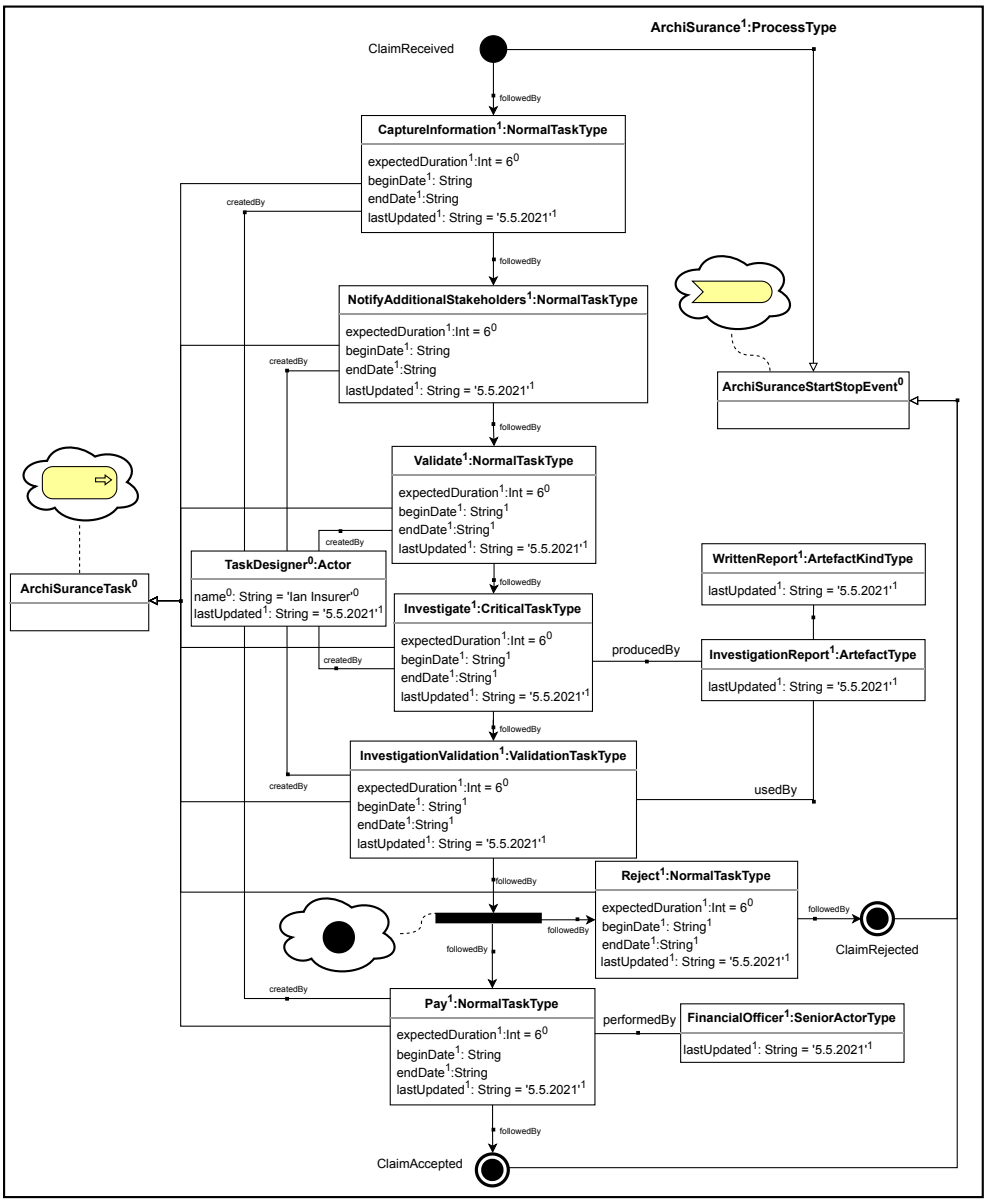


Figure 8: ArchiSurance Handle Claim Process

the domain specific notation(s) at the touch of a button. Also, new domain specific symbols can be added as needed at any ontological level.

In the case of Archisurance process types such as *Handle Claim*, it is likely that employees of Archisurance may sometimes wish to view process type descriptions using the ArchiMate concrete syntax illustrated in Fig. 7. Obviously, one way of achieving this would be to simply change the domain specific symbols assigned to the general process modeling language at O_1 to those of the ArchiMate concrete syntax (i. e., to change Fig. 6 to contain the ArchiMate symbols). However, this would change the domain specific symbols available to all users of the general process modeling language at O_0 (e. g., the ACME software engineering company in the second scenario) which is probably not desired. Many metamodels serve

as normative standards, and therefore users of the standard do not usually have the freedom to change the domain specific notation defined at the O_0 level to their own preference.

The advantage of the deep approach to domain specific concrete syntax definition, which exploits the underlying *OCA*, is that domain specific symbols can be added at any ontological level and are immediately available for visualization at that level or any level below. Moreover, the inheritance hierarchy structure can be used to avoid assigning the symbol multiple times. Fig. 8 provides an example of the use of this feature, where ArchiMate modelers have assigned ArchiMate-specific concrete syntax to certain elements in the *Handle Claim* process model. Since these assignments are performed at the O_1 level, they are local to the Archisurance context and do not affect the ability of other users of the O_0 model to use the more general concrete syntax defined in Fig. 6. After these new ArchiMate-specific symbols have been assigned to specific model elements of the *Handle Claim* process model, not only can they be used immediately to visualize those elements at the O_1 level, they are available to visualize instances of those model elements at the level below.

5.3 A Handle Claim Process Enactment

To complete the Archisurance Handle Claim example, in this section we show how a specific enactment of the Handle Claim process type, that processes a specific claim, can be represented at the O_2 level using the various concrete syntax options available (i. e., the general purpose *LML* syntax, the domain-specific process modeling language syntax defined at O_0 and the ArchiMate syntax defined at O_1). Since the Archisurance example does not contain any instances (i. e., enactments) of the Handle Claim process (since it cannot model them) we have created our own fictitious example for a particular claim (Claim 26152).

Fig. 9 shows the *Claim 26152* enactment of the Handle Claim process depicted using a mix of the standard *LML* notation and the O_0 domain-specific notation, while Fig. 10 shows the same process instance using the standard *LML* notation and the O_1 ArchiMate syntax added in Fig. 8.

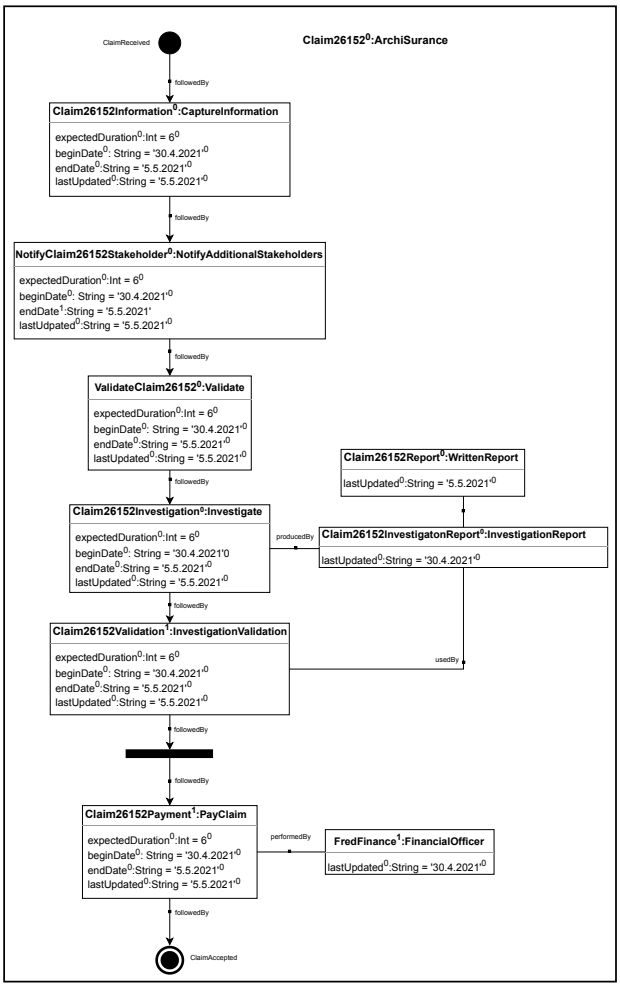


Figure 9: Handle Claim Process Enactment - LML *and general DSL Syntax*

6 ACME Application

This section considers the mandatory domain example from the Challenge for which additional domain-specific requirements are defined. The goal of this scenario is to define a concrete software engineering process for a fictitious software engineering company, the ACME Software Engineering Company, based on the general process modeling language defined in Sect. 4. As in that section, we first present the *LML* model and then discuss the domain specific requirements defined

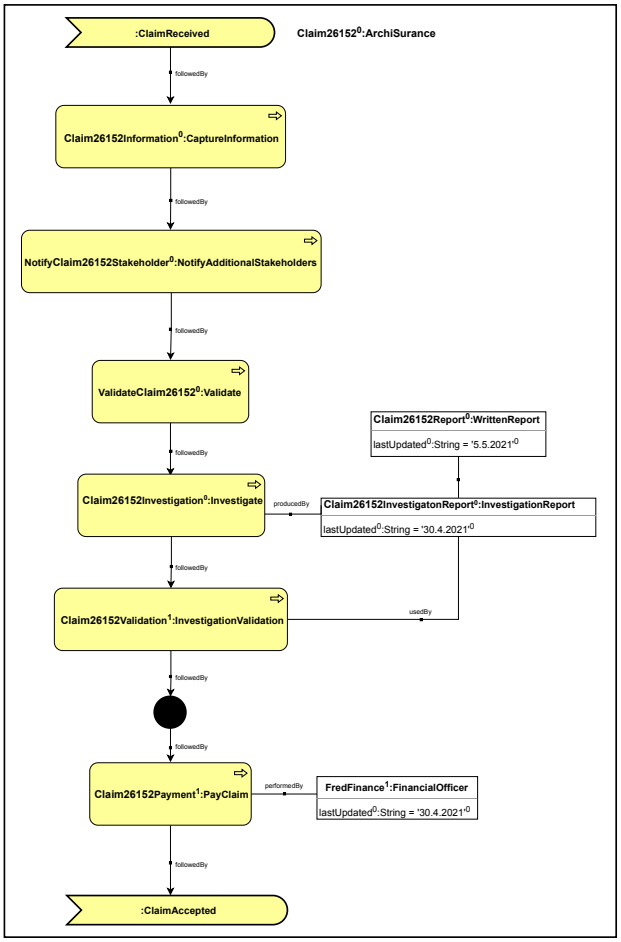


Figure 10: Handle Claim Process Enactment - LML *and ArchiMate Syntax*

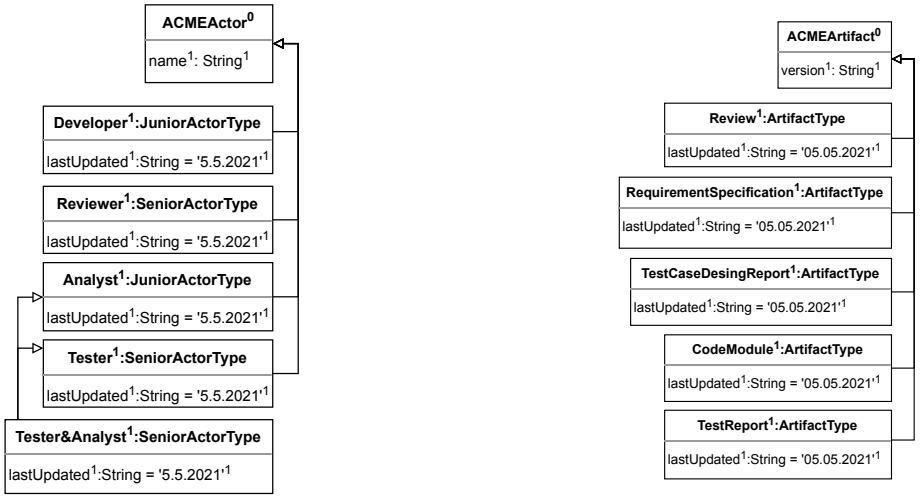
by the Challenge along with the additional *DOCL* constraints needed to support them.

6.1 ACME SE Process Description

The ACME Software Engineering (SE) Process is represented as an instance of the O_0 level general process modeling language defined in Sect. 4. According to the rules of deep modeling, therefore, it occupies the lower (in our case O_1) ontological level, and satisfies all the rules (i. e., requirements) that it defines (i. e., according to the Challenge description). Although the description of the process takes the form of a single, coherent model, for clarity purposes, we portray it using two separate views shown in Figures 11 and 12. The majority of clabjects appearing in the former also appear in the latter, but in both cases, classes with the same name represent the same model element. This follows the well-established UML convention that identically-named model elements, of the same kind (i. e., instantiated from the same metamodel element), represent the same model element. It would be possible to show the information in Figures 11 and 12 in one figure, but this would be much more cluttered.

Fig. 11 focuses on the specific `ArtifactTypes` and `ActorTypes` comprising the ACME SE Process and describes their relationships in terms of generalization sets. The generalization set for ACMEActor shows that there are five specific `ActorTypes` in the process, Developer, Reviewer, Analyst, Tester and Tester&Analyst, and that the latter is a specialization of both Analyst and Tester. This means that an instance of Tester&Analyst can serve as (i. e., play the role of) an instance of Analyst or Tester. The generalization set for ACMEArtifact, which shows that there are five specific Artifact Types in the process, Review, RequirementSpecification, TestCaseDesignReport, CodeModule and TestReport, plays two important roles. First, it clarifies that all specific ArtifactTypes in the model have a *version* attribute and second, it allows the power type pattern to be applied. In other words, it allows `ActorType` to be declared as the powertype of ACMEActor. As explained in Sect. 6.2 below, this is how the ACME SE Process ensures that any `ActorType` added in the future must have the same properties as the current `ActorTypes` represented in the model.

Fig. 12 focuses on describing the structure of the ACME SE Process in terms of the ordering constraints between the specific `Task Types` appearing in the process, as well as their relationship to all specific `Actor Types` and `Artifact Types` shown in Fig. 11. It also shows the specific Actor responsible for their design. This follows the basic structure of the process illustrated in Fig. 5 of the process Challenge. The process starts with the RequirementsAnaysis task type and then splits up the enactment of the process with an instance of AndSplit. The left hand side of the separated flow takes



(a) Actor powertype *(b) Artifact powertype*

Figure 11: Powertype definition in Level O_1

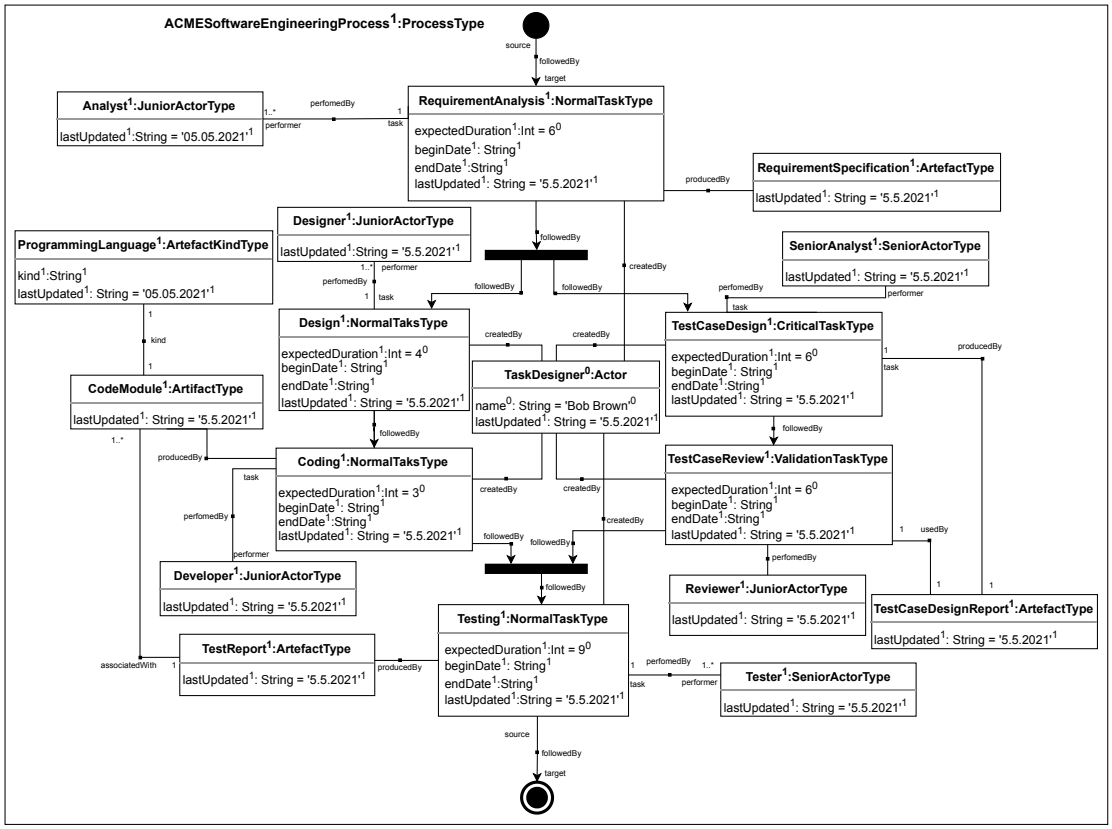


Figure 12: Task Type Flow in the ACME SE Process

care of the Design and the Coding tasks of the process. One or more instances of Designer perform the Design task and Coding is performed by one or more Developers. The right hand side introduces

the TestCaseDesign as a which is performed by a SeniorAnalyst which in turn is an instance of SeniorActorType. It also produces a TestCaseDesignReport which is used in the TestCaseReview task which follows the TestCaseDesign as an instance of `ValidationTaskType`. This task is performed by a Reviewer. The AndJoin that re-connects the flow of the process is followed by the Testing task which produces a TestReport artifact and is performed by a Tester. The TestReport artifact is then associated with another artifact which is CodeModule. Due to the fact that the Coding task can produce multiple CodeModules, the multiplicity constraint on the *associatedWith* connection is '1' on the TestReport end and '1..*' on the CodeModule end. Every CodeModule has to reference the programming language it is written in which is captured by the `ArtifactKindType` instance, ProgrammingLanguage, with the *kind* attribute. All instances of `TaskType` are created by the TaskDesigner whose name is"Bob Brown". This is an instance of Actor at level O_0.

6.2 Fulfillment of the Requirements

The authors of the Challenge formulated 13 domain-specific requirements for the ACME software engineering process. However, one requirement, S2, is explicitly flagged as being overridden by another requirement, S13, leaving only 12 active requirements. In this section, we explain how these requirements have been fulfilled and introduce the required *DOCL* constraints where necessary.

In order to enforce the powertype pattern for any future instances of `ArtifactType` and `ActorType` we have made use of the *DOCL* capability to define constraints on any level of a deep model. Constraint *powerArtifact* ensures that every instance of `ArtifactType` at the level O_1 of the ACME SE Process application has to be connected via an inheritance relationship to ACMEArtifact as a subclass. In combination with the invariant constraint called *powerACMEArtifact*, where ACMEArtifact is used as the context to make sure that every subclass has to be of type `ArtifactType`, this prevents any untyped or wrongly typed clabject participating as a subclass in this generalization set.

```
context O1(1,1)
inv powerArtifact: Clabject -> select(c|c.
    isDeepTypeOf(ArtifactType)) -> forAll(c|
    c.#getSupertypes()# -> includes(
    ACMEArtifact))
```

```
context ACMEArtifact(1,1)
inv powerACMEArtifact: self.#getSubtypes()#
    -> forAll(c|c.isDeepTypeOf(ArtifactType)
    )
```

The invariant constraints *powerActor* and *powerACMEActor* deal with the same problem as the constraints introduced for the `ActorType` power type pattern but instead of using the *isDeepTypeOf* operation, these constraints use the *isDeepKinfOf* operation since `ActorType` is an impotent clabject and is not able to produce direct offspring.

```
context O1(1,1)
inv powerActor: Clabject -> select(c|c.
    isDeepKindOf(ActorType)) -> forAll(c|c.#
    getSupertypes()# -> includes(ACMEActor))
```

```
context ACMEActor(1,1)
inv powerACMEActor: self.#getSubtypes()# ->
    forAll(c|c.isDeepKindOf(ActorType))
```

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

This requirement is captured by the *performedBy* connection between RequirementsAnalysis and Analyst as well as the *producedBy* relationship between RequirementsAnalysis and RequirementsSpecification. The multiplicity constraint on the connection ensures that a RequirementAnalysis is only performed by one or more Analysts.

S2) Overridden.

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

The first part of this requirement is fulfilled by the *performedBy* connection between *Coding* and Developer as well as the *producedBy* relationship between CodeModule and Coding. The second part is fulfilled by the *kind* relationship between CodeModule and ProgrammingLanguage with a multiplicity constraint of '1' on both connection ends.

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

This is fulfilled by the *kind* connection between *CodeModule* and *ProgrammingLanguage*.

S5) Coding in COBOL always produces COBOL code:

This requirement only makes sense for a coding task that only involves one programming language. If the CodeModule is connected to a ProgrammingLanguage via the *kind* relationship and it references COBOL then the language of the CodeModule is COBOL.

S6) All COBOL code is written in COBOL:

This requirement is again fulfilled by the *kind* connection between CodeModule and ProgrammingLanguage. By reifying the notion of programming language, and representing the language in which a programming language is written by means of a connection to a programming language object rather than by a String attribute, this requirement is automatically fulfilled by the creation of the connection.

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

The first part of this constraint says that *Ann Smith* is a Developer which means that at level O_2 an instance of Developer has to exist with the name *Ann Smith*. This is captured by the following constraint

```
context Developer(1,1)
inv AnnSmith: self.allInstances() ->
    exists(d|d.name = 'Ann Smith')
```

The second part of this statement says that if a coding task exists that uses 'COBOL' as the programming language the only actor that can perform this task is *Ann Smith* which has to be a Developer. The fact that *Ann Smith*, as a Devloper, can only be connected with tasks that are instances of Coding tasks is ensured by the *performedBy* connection.

```
context ProgrammingLanguage (2,2)
inv: self.kind = 'COBOL' implies self.
    CodingModule.Coding.performer = 'Ann
    Smith'
```

The constraint defined in *ProgrammingLanguage* is only evaluated at level O_2.

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

This requirement is again fulfilled by the *performedBy* connection between Testing and Tester and between Testing and TestReport.

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

This requirement is fulfilled by the *associatedWith* connection between CodeModule and TestReport

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 first part of this requirement is fulfilled by assigning actors to the respective tasks they can perform. Due to the general typing restrictions on connections, it is impossible to introduce new connections that connect a Developer to TestCaseDesign, for example. From the produced artifact that is connected to the task that, in turn, is connected to the actor that performs

the task, we can determine the responsible actor implicitly.
The second part of this requirement is fulfilled by including the generalization sets for ACMEActor and ACMEArtifcat and using these to indicate, through inheritance, that all specific `ArtifactTypes` have a version attribute and all ACMEActors have a name.
The third part of this requirement, stipulating that new artifact types added to future versions of the ACME SE Process must also satisfy the first part of the requirement, is fulfilled by declaring the clabject `TaskType` at the O_0 level to be the powertype of ACMEArtifact at the O_1 level. This ensures that every future instance of `TaskType` is a subclass of ACMEArtifact and thus has the required version attribute and connection to ACMEActor. The constraint to enforce this power type pattern was defined at the beginning of Sect. 6.2.

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

The first part of this requirement is fulfilled by the following constraint.

```
context Tester&Analyst(1,1)
inv BobBrown: self.allInstances() ->
    exists(d|d.name = 'Bob Brown')
```

The second part is fulfilled by the *createdBy* relationship between TaskDesigner and all the instances of `TaskType` in the model.

S12) The expected duration of testing is 9 days:

This requirement is fulfilled by the fact that the value of the attribute *expectedDuration* in *Testing* at level O_1 is set to '9' (which means 9 days) and the fact that the vitality property 'Mutability' is set to the value '0'. This means that the *expectedDuration* attribute of instances of *Testing* must have that same value.

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

This requirement is captured by the fact that the clabject TestCaseDesign is an instance of `CriticalTaskType`, and by the *performedBy* connection between TestCaseDesign and SeniorAnalyst which is an instance of `SeniorActorType`.

6.3 An ACME SE Process Enactment

The Challenge requests that an example enactment of the ACME SE process should be included to illustrate how they would be handled, but does not define any particular requirements that have to be fulfilled. In this section, we, therefore, present an example enactment of the ACME SE process called Simple System to indicate that it was used to generate a software system of the same name. This enactment, illustrated in Fig. 13, is an instance of the previously presented ACME SE process model, and, therefore, occupies the O_2 level of the deep model according to the rules of strict modeling.

Since the ACME SE process has no or-splits nor or-joins, the basic content and layout of the model is similar to the O_1 level. Basically, every task type in the O_1 level has an instance at the O_2 level with the corresponding *followedBy* connection. The main difference is that this enactment model identifies the actual indirect instances of ACMEActor that carried out each task. For example, *Ann Smith* performed the Coding task called *SSCoding*. This also compels every instance to have a *name* attribute. Every indirect ACMEArtifact instance also has to have a *version* attribute due to the power type pattern enforced at the level above (O_1).

7 Discussion

The solutions presented in this paper fulfill all the mandatory requirements in the sense that all the required expressivity is enabled, and all the prohibited expressions are forbidden. However,

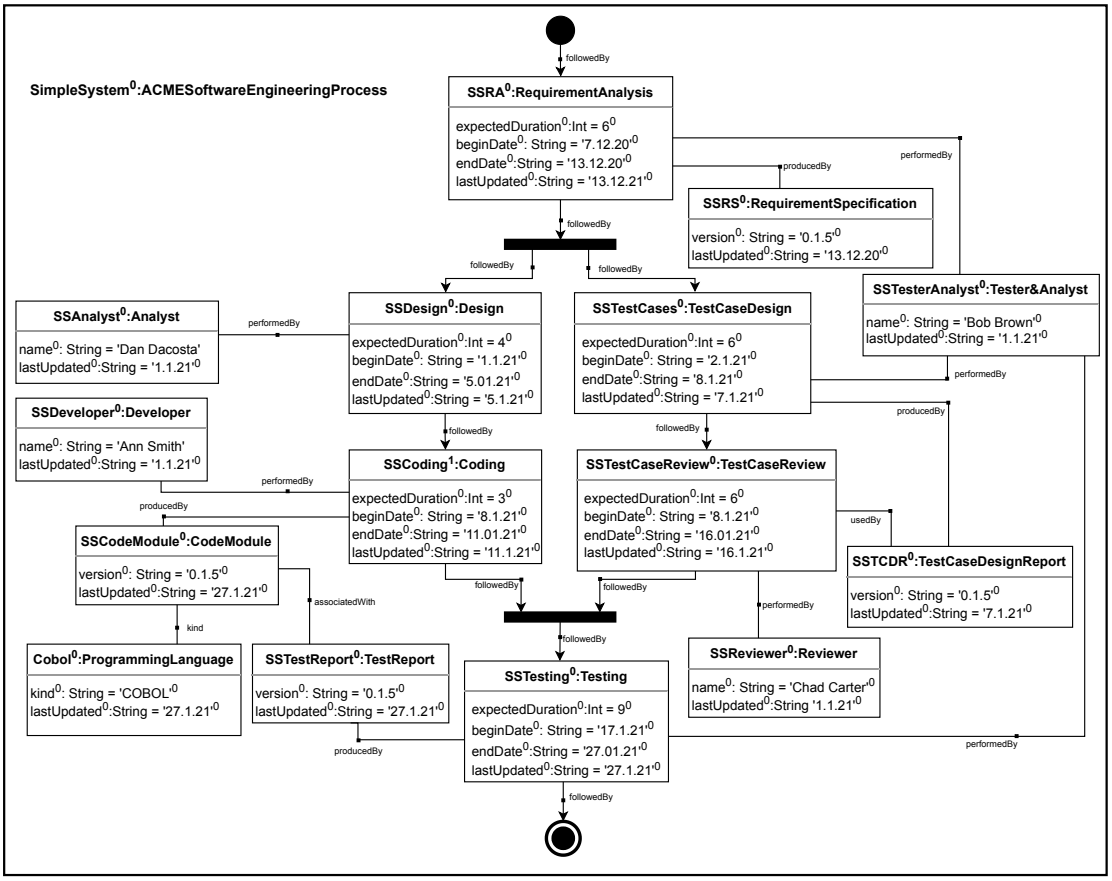


Figure 13: Level O_2

not all the explicit or implied non-functional requirements may have been solved optimally or in precisely the way implied. In this section, we discuss these issues and address all the discussion points (mandatory and optional) defined in the Challenge.

7.1 Mandatory Discussion Points

This subsection first addresses the mandatory discussion points.

Basic Modeling Constructs

The basic modeling constructs used in the solution were explained in the introduction to *LML* and the deep modeling approach it supports in Sect. 2. As mentioned there, *LML* was designed with the goal of supporting the look-and-feel of the UML class diagram notation using the smallest possible number of level-agnostic concepts (i. e., abstract syntax elements) and notational symbols (i. e., concrete syntax elements). For example, as well as the unified representation of clabjects, there are no representational differences between attribute definitions and attribute instances (cf. slots), since these are regarded as special cases of the more general notion of "deep attributes". In other words, the *LML* takes every opportunity to obviate superfluous UML modeling features which add additional notational complexity without enhancing the language's expressive power. Examples include:

- obviating stereotypes, tags/tagged values, extensions and related concepts, since the same information can be expressed more succinctly using specialization relationships, deep attributes and vitality properties,

- obviating association classes and the complex "end" structures of associations by combining associations/links into the unifying concept of "connections". Connections are clabjects that have a level agnostic structure/notation and, when desired, can be treated as classes/objects in their own right (like association classes/instances),
- obviating the need for special labels for model elements with specific kinds of type/instance properties using the vitality properties, especially potency. For example, abstract classes are represented as superclasses with potency zero, while dependencies are represented as impotent (i. e., potency zero) connections between potent (i. e., potency one or higher) entity clabjects.

Levels

Since the *LML* modeling language is based on the tenets of strict modeling, the levels in our solution are determined by classification levels. Thus, all the ontological instances of the top O_0 level reside at O_1 and all the instances of O_1 reside at O_2 in both scenarios (i. e., software engineering processes and insurance claim handling processes). However, in contrast to some multi-level modeling approaches, there is no requirement that the potencies of clabjects match their level. This is the case, for example, in the MetaDepth approach (De Lara and Guerra 2010), where the potencies of clabjects are always the same as the level they occupy. This is possible because MetaDepth, like most *MLM* approaches, adopts the UML infrastructure's numbering scheme in which the most concrete classification level is assigned the number zero, and the level containing the types of the bottom level is assigned level one and so on. In *LML* models, including the example deep models in this paper, all of the vitality properties (potency, durability, and mutability) are only related to the level number in one way – they cannot be larger than the total number of ontological levels in the deep model.

Number of Levels

Both solutions presented in this paper are comprised of three levels. This reflects the fact that the Challenge description essentially defines three different characterization contexts for each application:

- the context of characterizing (i. e., defining the concepts appearing in) the general domain of modeling processes. This is common to both application domains,
- the context of characterizing (i. e., defining the concepts appearing in) process enactments in a specific application of the aforementioned context - one in the domain of software engineering and one in the domain of insurance claim handling,
- the context of representing concrete process enactments of the aforementioned context for each of the two examples.

Each of these contexts maps naturally to an independent level, with each (except the first) adding relevant information based on the constraints and expression possibilities introduced by the preceding context. Had the challenge added additional, intermediate characterization contexts, such as describing the general properties of all software engineering processes, for example, these could have been mapped to additional intermediate levels.

Cross-Level Relationships

As mentioned in Section 2.1, strict modeling allows only one kind of relationship to cross levels, the classification relationship, also known as the *instanceOf* relationships. This is simultaneously one of the biggest strengths and weaknesses of the strict modeling tenet upon which the presented solutions are based. It is a strength because it provides concrete guidance about what levels are needed to fully characterize a domain and what content those levels should have, and in many situations is relatively simple and straightforward to apply. It is a weakness, however, because in certain situations, namely when a domain involves two or more characterization chains (i. e., classification

backbones) which do not naturally align, superfluous model elements and/or inelegant workarounds are needed to maintain strictness. Potential ways of addressing this issue are discussed in Sect. 8.

Deep Characterization

The main way *LML* itself supports deep characterization is by means of its three vitality properties - potency, durability, and mutability. The potency value of '2' for many of the concrete clabjects in the O_0 level ensures that the concepts they capture may be instantiated not only at the O_1 level but also at the O_2 level. When combined with suitable constraints, as discussed below, it is also possible to require certain instances to exist at certain specific levels below.

Durability captures a similar notion to potency, but for the features of a clabject (i. e., attributes and methods) rather than the clabjects themselves. Several examples of durability '1' and durability '2' attributes can be seen in the O_1 level model, such as the *lastUpdated* attribute of `Element`, which has a durability of '2'. This is higher than the potency of the clabject, which at first sight might seem counterintuitive. However, note that the *lastUpdated* attribute is inherited by the concrete subclasses of `Element`, and so is able to deeply characterize its indirect instances.

In contrast to potency and durability, mutability places no requirements on the existence of clabjects and attributes at a lower level, it places constraints on where (i. e., at what levels) the values of attributes that do exist can be changed. For example, all the `TaskType` clabjects in the model of the ArchiSurance Handle Claim Process in Fig. 8 have an *expectedDuration* attribute with a durability of '1', a mutability of '0' and a value of '6'. This means that all instances of these task types also have to have an *expectedDuration* attribute with the value '6'.

Cross-Level Constraints

The *DOCL* extension to *OCL* used in the presented solutions was explicitly designed to support the definition of constraints that can recognize, and operate over, both classification dimensions, and in the case of the ontological dimension, over an unlimited range of levels. For example, the majority of the constraints defined in Sect. 4.2 have a context modifier that ends with the expression (1,2). One of the constraints of this form from Sect. 4.2, which helps to fulfill Requirement P2, is shown below.

```
context TaskType(1,2)
inv taskFollowedBy: self.source -> size() =
    1 and self.target -> size() = 1
```

The basic purpose of this constraint is to ensure that instances of `TaskType` must be involved in two *followedBy* connections, one in which it plays the role of the source and one in which it plays the role of the target. The effect of the expression (1,2) after the name of context clabject (i. e., `TaskType`) is to ensure that not only immediate instances of `TaskType` have to satisfy this requirement, but also the instances of those instances. The *taskFollowedBy* constraint is, therefore, a "deep" constraint which applies to, and is checked at, the two levels immediately below the context clabject.

Integrity Mechanisms

The *Melanee* tool used to model the solutions presented in this paper offers two automated integrity checking mechanisms. The first, and most powerful, is the *DOCL* dialect which is integrated into the *Melanee* tool. This can be invoked at any time to check whether the deep model as a whole adheres to all *DOCL* invariant constraints. For example, when asked to check the above deep *DOCL* invariant the tool will visit all deep instances of `TaskType` at both the O_1 and O_2 levels to ensure that they have the required number of *followedBy* connections in which they participate in the correct roles. Because the constraint is a deep constraint which applies to the two levels below that in which it is defined, all instances of `TaskType` at both the O_1 and O_2 levels will be checked.

The second integrity mechanism is the so-called emendation service which checks whether the vitality properties of the clabjects in a deep model

are all consistent with one another and satisfy the rules. The mechanism is automatically invoked whenever a change is made to a deep model, at any ontological level. If the tool detects an inconsistency, it flags the issue and offers the modeler some options for correcting the problem.

Generality

Both solutions for both of the example applications have been carefully designed so that the requirements and properties defined in the different characterization contexts of the Challenge are separated and handled at different levels. Thus, all the rules and requirements defined in Sect. 2.2 of the Challenge description that relate to the generic properties of process models, including domain specific concrete syntax, are captured completely and exclusively in the O_0 level *LML* model and the accompanying *DOCL* constraints (Sect. 4). Similarly, all the rules and requirements for the example processes for the two application domains, including domain specific concrete syntax in the Archisurance case, are captured completely and exclusively in their respective O_1 level models (Sect. 5 and 6). Finally, all the properties and information about the example enactments of these two processes are captured completely and exclusively in their respective O_2 level models (Sect. 5.3 and 6.3).

The separation of concerns in this way means that the description of each characterization context is as general and reusable as possible. In particular, since the description of the generic rules are self contained at the O_0 level, this level can be reused (i. e., instantiated) as many times as desired without each new instance (i. e., application) affecting any of the existing applications. Similarly, since the descriptions of the example processes in the two application domains are self contained at the O_1 level, they can be enacted (i. e., instantiated) as many times as desired without each new instance (i. e., enactment) affecting any of the existing enactments.

Extensibility

Melanee is a fully data (i. e., model)-driven tool which treats all ontological levels equally and allows any model element at any level to be changed at any time, with the changes taking immediate effect. Moreover, since it is built around the time honored object-oriented modeling mechanisms of inheritance and instantiation, any model at any level can be extended with more subclasses and/or instances. Fundamentally, therefore, *LML* models created using *Melanee* are highly extensible. The key challenge is controlling all the possibilities for making extensions.

Although it has not been mentioned thus far in the paper, *Melanee* supports another time honored object-oriented modeling feature for controlling and organizing extensions – packages. Packages are restricted to containing elements within a given ontological level, but otherwise, they can be used just like UML packages to group related model elements. As mentioned previously, dependencies between packages can be modeled as impotent connections. The other important mechanism for ensuring that extensions are controlled is the ability to explicitly enforce the use of the powertype pattern through *DOCL* constraints, as shown in Sect. 6.2. This can be used to ensure that new model elements added to a particular ontological level in the future must have certain properties and relationships. For example, the use of the powertype pattern in the ACME SE process ensures all ACME actors added in the future must be instances of the O_0 level `ActorTypes` clabject, and that all instances of `ActorTypes` added in the future must be subclasses of the O_1 level ACMEActor clabject.

7.2 Recommended Discussion Points

This section briefly addresses the two recommended, but not mandatory, discussion points.

Formalisms

A set-theoretic semantics for the core concepts underpinning *LML*'s flavor of deep modeling was defined by Kennel (2012), while the semantics of *DOCL* is based on first-order logic like the *OCL* language it extends. As with *OCL*, *DOCL* can

not be used as a standalone modeling language but has to be used in combination with an underlying modeling language (in our case, *LML*). Like OCL, *DOCL* constraints are used to enhance the precision of models with extra information. However, in contrast to standard *OCL* which is purely declarative, *DOCL* constraints are also executed on models in order to calculate values for attributes enhanced with a *derive* or *init* constraint or to check if the current status of a model violates any invariants.

Verification

As mentioned previously, one of the main integrity checking mechanisms supported by the *Melanee* tool is the *DOCL* constraint checking engine which is an extension of the OCL. In other words, *DOCL* supports the full range of *OCL* features within, and over, ontological levels, and provides the aforementioned extensions to generalize them over multiple levels. Moreover, since *Melanee* is based on the Eclipse Modeling Framework (EMF), the full range of EMF language and tools can be used on *LML* models, albeit in a level-unaware way.

To assess our solution quantitatively, we applied the 'Control Capacity' metrics described in Kühne and Lange (2020) to the 'ACME Software Engineering Process' solution. This metric is used to describe the level of control the type levels excerpts over the instances created in the levels below. It indicates a certain degree of well-formedness of a model and ensures that types are actually used at the level below to ensure that it has the intended structure. There are two dimensions to this metric, the subtyping dimension, and the classification dimension. These are the only ways clabjects can assert control over other clabjects.

$$CC = \frac{1}{|C| * (1 + tcw)} \left(\begin{array}{c} \sum_{e \in C_s \perp} ansc\_s(e) \\ + \\ tcw * \sum_{e \in C_t \perp} ansc\_t(e) \end{array} \right)$$

The expressions inside the bracket calculate the number of ancestors in the subtyping and classification dimension and aggregate their respective values. The term *tcw* is a weight value to balance the influence of the subtyping versus the classification dimension, which is then incremented and multiplied by the number of clabjects in the model.

When evaluated on our ACME SE Process solution this equation gives a control value of *0,956* which is very high. To ensure this level of control is actually maintained, the equation can be translated into an invariant *DOCL* constraint on a deep model (i. e., the context of the constraint is the *DeepModel*).

8 Systematically Relaxing Strictness

As mentioned previously, the principle of strict modeling is both a strength and a weakness of multi-level modeling approaches. It is a strength because it provides a basic structuring principle for multi-level models without which the notion of well-defined levels starts to dissolve, and it is a weakness because many domains have concepts and relationships that cannot all be modeled in the most natural way within a strict modeling scheme. The role of strict modeling is therefore one of the most discussed aspects of multi-level modeling (Eriksson et al. 2013) and many of the features of multi-level modeling approaches have been proposed to try to overcome its restrictions. Examples include leap potency (De Lara and Guerra 2010), dual deep potency(Neumayr et al. 2018) and join potency (Theisz et al. 2020).

The version of deep modeling supported by *LML* and *Melanee* is one of the few *MLM* approaches that still applies the principles of strict modeling. The price, however, is that sometimes scenarios in a domain of interest cannot be modeled in the most natural way, and special workarounds have to be used to reconcile them with strict modeling. Our solution to the ACME software engineering process uses such a workaround to align the characterization hierarchy of actors with the characterization hierarchy of tasks, which are created and performed by actors. While this does not invalidate the claim that the solutions fulfill all requirements, it does detract from

the approach's overall goal of intuitiveness and reducing accidental complexity, which is a weakness. In this section, we, therefore, describe the problem in more detail, explain the workaround used to address it in our solution, and discuss ways in which better solutions could be provided in the future.

8.1 Aligning Incongruent Classification Hierarchies

It is not by accident that the Challenge includes a situation in which a single human actor, in this case, *Bob Brown*, plays different roles in relation to two clabjects that naturally occupy different levels in the deep model. In the Challenge requirements, *Bob Brown* is on the one hand described as being the designer for all the task types in the ACME Software Engineering process, and on the other hand as being the tester/analysts responsible for performing specific instances of those task types (e. g., testing and analysis tasks). This immediately creates a conundrum about the location of the model element representing *Bob Brown* since it is directly related to two clabjects that occupy different levels in the deep model.

Fig. 14 highlights the nature of this conundrum and how it was addressed in our solution. The classification hierarchy on the right-hand side of Fig. 14 shows the "task" classification hierarchy that forms the main backbone of the deep model, in which all the clabjects involved, `NormalTaskType`, Testing and *SSTesting*, have their natural position from a strict modeling point of view. The rest of the model shows the essential nature of the problem and the workaround we used to avoid it in our solution.

In the description of the ACME SE process, *Bob Brown* plays two explicit roles – the role of the designer of all the tasks (i. e., `TaskType` instances), such as a Testing, which occupy the O_1 level, and the role of a tester and analyst who performs instances of specific tasks, such as the *SSTesting* instance of the Testing task. If *Bob Brown* were represented most naturally as a single clabject in the model, it would have to have connections to two clabjects, Testing and *SSTesting*, that occupy different levels. The latter is in fact an instance of the former. This is not allowed in the strict modeling approach since one of the connections would have to cross a level boundary.

The workaround we use in our solution to get around this problem is to avoid modeling *Bob Brown* directly as a single clabject, but instead to model the roles that he plays (i. e., TaskDesigner and *SSTester* roles) and to relate them to *Bob Brown* in an indirect way using their name attribute. As shown by the two clabjects with a red boundary in Fig. 14, this essentially means that *Bob Brown* is actually being represented twice in the model, once in his role as task designer and once in his role as tester and analyst. The use of this "trick" is unlikely to cause confusion with software engineers and provides all the information needed to build a system that fulfills the requirements, so it technically fulfills the Challenge. However, as highlighted by Fig. 14, many modelers would regard this as an unnatural way of modeling this situation for at least two reasons. Not only are there two clabjects in the model which represent the same real-world object, the natural intention behind the use of the terms Actor and `ActorType` in the Challenge description is that the occurrences of the former are intended to be instances of the latter. In other words, *Bob Brown*, in his role as TaskDesigner is naturally an instance of the clabject `SeniorActorType` in the top left of the figure, rather than an instance of the class Actor which we have added to serve as the type of TaskDesigner. The use of the workaround, therefore, involves two modeling choices that most experienced UML modelers would probably consider unnatural.

8.2 Accommodating Different Multiple Modeling Spaces

The underlying cause of this problem, and many others like it, is that deep models sometimes have multiple "backbone" classification hierarchies, or dimensions of concern, where clabjects in one hierarchy are related to multiple clabjects at different levels of another hierarchy. Atkinson and Kühne (2001a) referred to these different classification

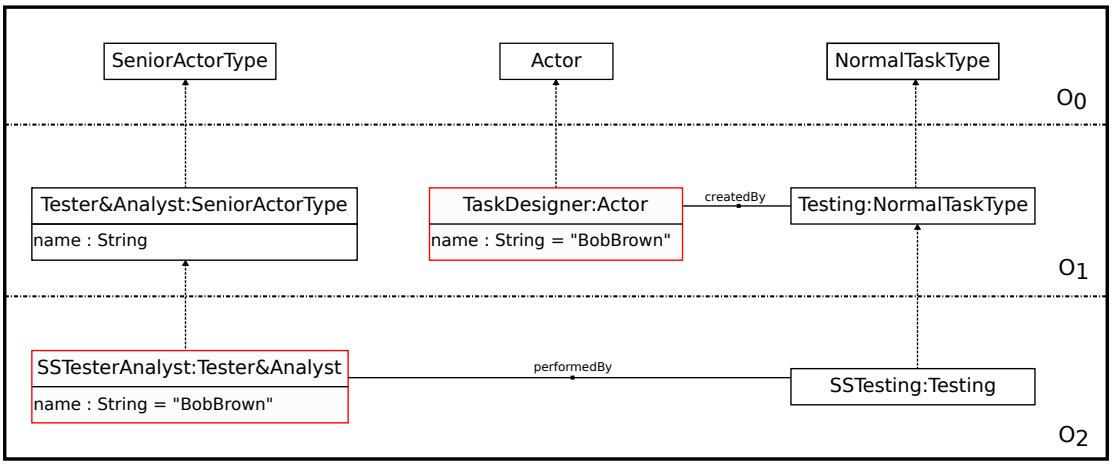


Figure 14: Workaround Used in the ACME SE Process Solution

"backbones" as metamodeling spaces and recognized that a potential way of balancing the goals of strictness and flexibility was to systematically and explicitly separate these metamodeling spaces[2] and handle the relationships between them in a special way. Fig. 15 shows, schematically, how this could be achieved in the case of the *Bob Brown* scenario.

Essentially, Fig. 15 separates the "task" classification hierarchy on the right-hand side of Fig. 14 from the rest of the model and places the clabjects it contains into their own modeling space called "Task Space". The rest of the model focuses on describing the "actor" classification hierarchy, but in a "natural" way that does not have to worry about the level-crossing connections described above. Freed from the "task" classification hierarchy, in the "Actor Space", *Bob Brown* can be represented as a single clabject that is an instance of the clabject Person, and all the different actor types can be placed in a natural hierarchy based on their ontological instantiation relationship to `SeniorActorType`. Having separated the two main classification backbones (the Actor Space and the Task Space) into their own modeling spaces, the relationships between clabjects in different spaces can be represented as needed using special dimension-crossing relationships which do not have to worry about ontological classification levels. Thus, as shown in Fig. 15, TaskDesigner can be connected to `NormalTaskType` as the designer of its instances, Tester&Analyst can be connected to Testing as the performer of its instances, *SSTaskDesigner* can be connected to Testing as its designer and *SSTester* can be connected to *SSTesting* as its performer. Each of these connections, which are represented in blue in Fig. 15, is a dimension-spanning connection that does not have to adhere to any notion of strictness. Within a modeling space (i. e., dimension), however, the rules of strictness have to be respected as before. This approach, therefore, provides a balance between complete strictness, where every model element in the whole model has to respect the rules of strictness and zero strictness, where no model elements of any kind have to respect the rules of strictness.

8.2.1 Interconnected Deep Models

Although the modeling spaces concept has been present before (Atkinson and Kühne 2001a), to our knowledge it has yet to be mapped to a set of pragmatic modeling features in a concrete tool. The 3D representations shown in Fig. 15 or in (Atkinson and Kühne 2001a) are helpful for visualizing the concept of modeling spaces, but

[2] In the rest of this paper we use the term "modeling space" rather than "metamodeling space" since the classification hierarchies concerned contain multiple ontological classification levels, not just the meta ontological level.

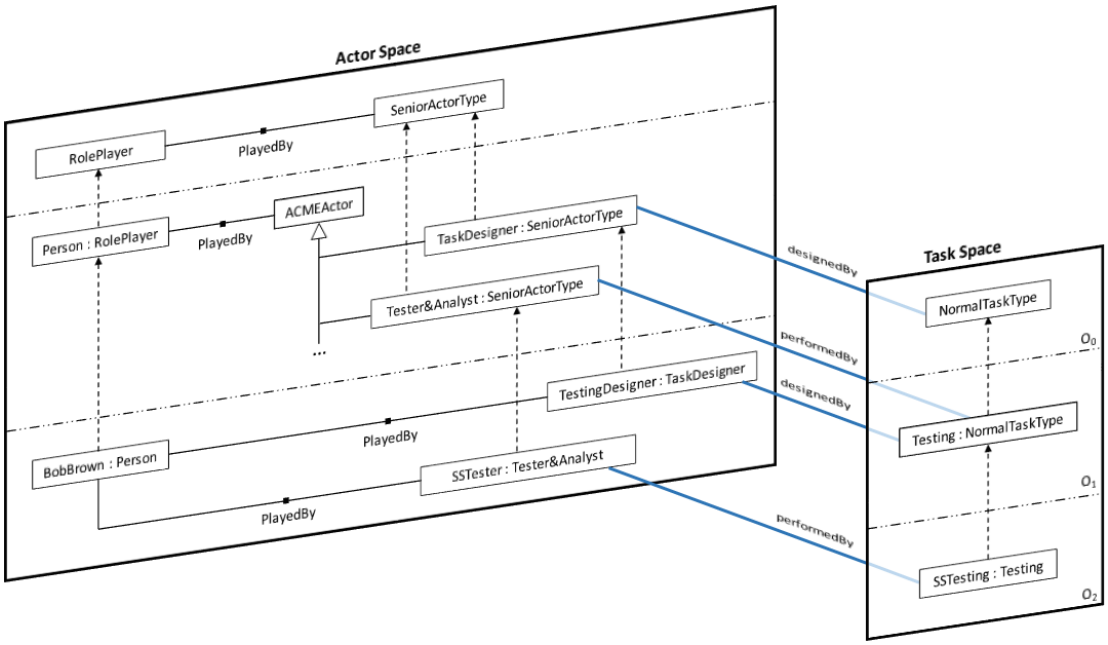


Figure 15: Schematic Representation of Modeling Spaces

they are not pragmatic solutions for a concrete modeling tool.

The *Melanee* environment used to model the solution presented in this paper does, however, potentially provide the foundation for a pragmatic way of allowing modelers to work with modeling spaces because it treats deep models, which are essentially modeling spaces, as first classes citizens of the linguistic metamodel. This can be seen in the top left-hand corner of Fig. 3, the linguistic metamodel underpinning the solution, where *DeepModel* is explicitly modeled as a class. The rationale for doing this in *Melanee* is that there should only be one deep model for a particular project or domain and that this deep model should be the top level container for all the levels which in turn are the containers of all the clabjects. However, it would be relatively straightforward to allow multiple deep models to be created for a given domain of interest, and to allow them to be nested within a higher level container, such as packages, which are also supported as first-class citizens in *Melanee*. If adapted in this way, the modeling space solution depicted schematically in Fig. 15 could be modeled concretely in the way shown in Fig. 16. This has the same basic arrangement of clabjects as in Fig. 15, but is represented much more pragmatically in terms of two deep models contained within a package. The benefit of such an approach is that the rules of strictness only have to be adhered to by the clabjects and connections within each deep model, but connections between deep models, which are now clearly distinguishable, would be governed by much more relaxed rules, if any. The "solution" for the fragment of the Challenge depicted in Fig. 16 manages to systematically represent the required relationships between the two metamodeling spaces at the same time as modeling the Actor Space in a way that is both natural and conformant to the rules of strict modeling.

8.3 Deep, View-Based Modeling

The approach shown in Fig. 16 essentially manages to relax the strict modeling rules in a systematic way by abandoning them at the "global" level while still enforcing them at a "local" level. However, modelers still have to identify the modeling spaces required and make rigid decisions about the locations of the clabjects within them. Perhaps

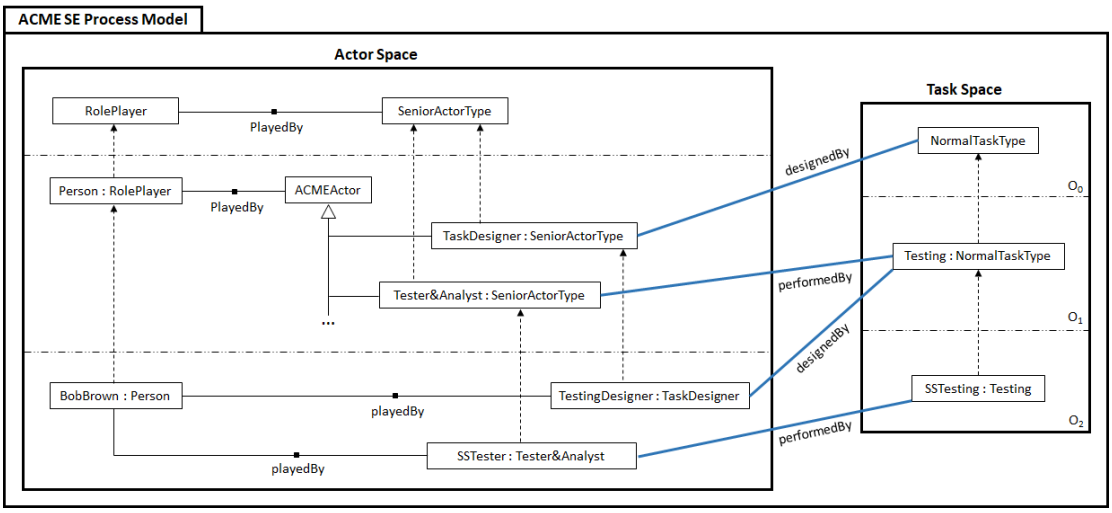


Figure 16: Nest Deep Model Representation of Modeling Spaces

the most powerful and flexible way of achieving a balance between global strictness rules and local strictness rules, therefore, is to move towards a fully view-based approach in which (locally) strict portrayals of clabjects are regarded as simply views of a larger, underlying pool of clabjects – a Single Underlying Model (SUM) – which is globally non-strict. Tunjic (2021) has developed such a deep, view-based modeling environment in which deep views are projected on demand from a deep SUM, but not for the goal of relaxing strictness rules systematically.

This approach is depicted in Fig. 17. The dashed oval in the top left of the figure represents the SUM for the domain (i. e., the application in our case) which captures all relevant information about the domain in a non-strict, but nevertheless in a disciplined way. For illustrative purposes, Fig. 17 simply shows the SUM as containing all the clabjects from Fig. 16, but without the modeling spaces. However, the notion of modeling spaces would no longer be needed. The rectangles surrounding this SUM represent views that are projected from the SUM, on-demand, to display a certain subset of the content of the SUM in a strict way. Again, for illustration purposes, one of the illustrated views, the "Task Hierarchy" view has been made to resemble the task modeling space from Fig. 16, but there is no need for the contents of views to be designed according to the modeling space concept. The only requirement is that the information within them be portrayed in a (locally) strict way. The "Player Hierarchy" view is similar to the "Task Hierarchy" but focuses on *Bob Brown*'s classification hierarchy. Finally, the two remaining views show the two roles that *Bob Brown* plays in the process. The "TestingDesigner Player" view shows that *Bob Brown* played the role of the designer of the Testing task, while the "SSTester Player" view shows that *Bob Brown* played the role of the *SSTester* in the *Simple System* enactment of the ACME SE Process.

Note that *Bob Brown* is represented by a single clabject in the SUM, but appears in various views to show his relationship to other clabjects as needed. Each appearance of *Bob Brown* in a view is consistent with the rules of strict modeling, although the network of connections the *Bob Brown* model element is involved in inside the SUM is not. This approach, therefore, satisfies the goal of relaxing strictness in the overall, global model (the SUM) while enforcing it locally within each view. The SUM does have to adhere to some rules, however. More specifically, for such an approach to work, the SUM would have to be free of metacycles (Atkinson and Kühne 2001a).

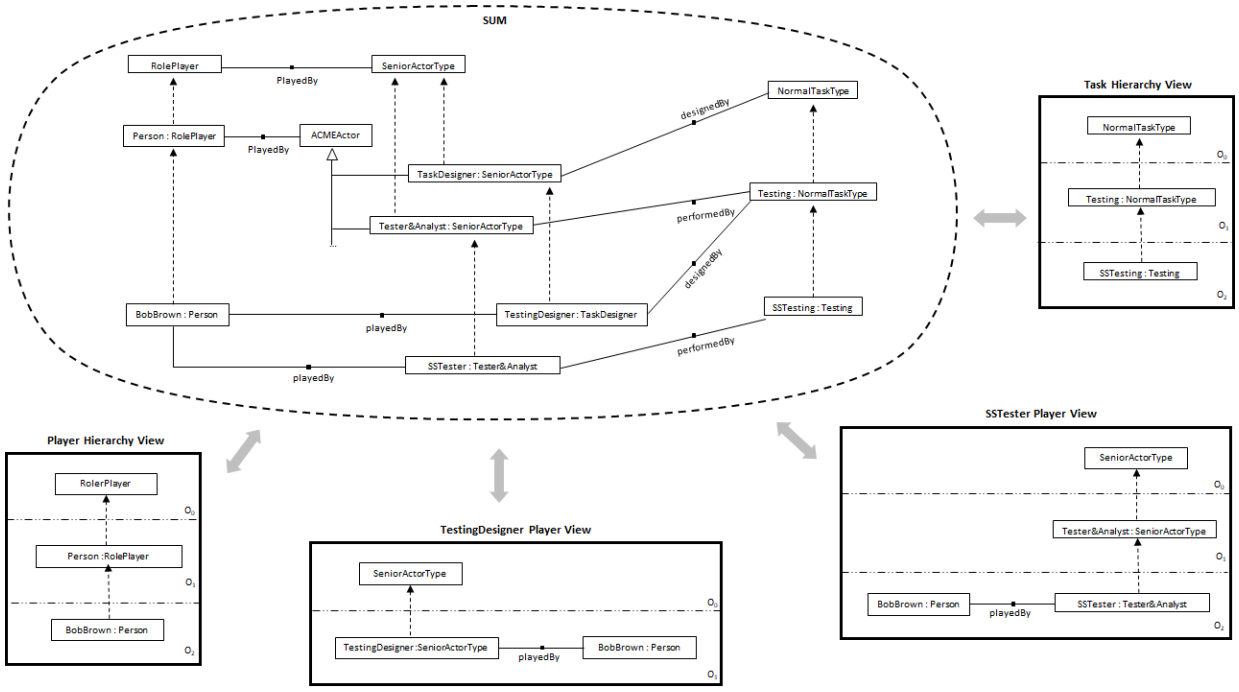


Figure 17: View-Based Attainment of Local Strictness

9 Related Work

The first modeling tools that treated ontological classification relationships as first classes citizens of models, and thus allowed rudimentary multi-levels models to be created, appeared in the early 1990s, such as ConceptBase (Jarke et al. 1995), Telos (Mylopoulos et al. 1990), and Ptolemy (Eker et al. 2003). However, these tools did not try to impose any rules on the kinds of ontological classification graphs that could be created, nor try to support a mainstream modeling notation such as the UML. The new generation of multi-level modeling tools and approaches, which are the focus of the MULTI workshop series and the Challenge this paper presents a response to, can be traced back to the turn of the Millennium when notions such as the orthogonal classification architecture, strict modeling, clabjects, and potency were proposed to ease UML modelers' struggles with linguistic meta modeling, profiles, powertypes and the application of the type-instance pattern (De Lara et al. 2014b). From these roots, a large variety of different multi-level modeling approaches have emerged, with a large variety of different features and abstraction mechanisms (Frank 2018). Nevertheless, they shared the basic goal of allowing ontological classification to be exploited in a more flexible yet controlled way (De Lara et al. 2014b).

A feature based categorization published at the MULTI 2016 workshop (Igamberdiev et al. 2016) identified 21 different multi-level model approaches. These range from strategies for using existing languages like the UML in a multi-level way (e. g., Power type metamodeling (Gonzalez-Perez and Henderson-Sellers 2006), VPM (Varró and Pataricza 2003), VMTS (Levendovszky et al. 2005)) or extending existing modeling platforms/tools to support *MLM* features (e. g., DPF Workbench (Lamo et al. 2012), *Melanee* (Atkinson and Gerbig 2016), OMME (Volz and Jablonski 2010)), to new tools/languages that support *MLM* from "the ground up" (e. g., MetaDepth (De Lara and Guerra 2010), XModeler (Clark and Frank 2020)) or apply *MLM* concepts in the context of an underlying formal system (e. g., MLT(Carvalho et al. 2015), M-Objects (Neumayr et al. 2009), Nivel

(Asikainen and Männistö 2009), Dual Deep Modeling (Neumayr et al. 2018) and FOML (Balaban et al. 2018)).

Furthermore, since that comparative study was published, several more *MLM* approaches have been described in the literature, such as MultEcore (Macías et al. 2018), FMMLx (Frank 2014), ML2(Fonseca et al. 2018), MLT-Telos (Jeusfeld et al. 2020), and DMLA (Somogyi et al. 2019). It is beyond the scope of this paper to attempt to compare all of these approaches and characterize the full range of concepts they support. Ideally, each of these approaches should be used to model the same scenario, such as this process Challenge or the MULTI 2018 Bicycle Challenge (Clark et al. 2018), to compare their respective pros and cons.

Recently an article was published (Kühne 2022) that introduced the mechanism of 'orthogonal ontological classification' which takes the idea of interconnected modeling spaces further. This new classification allows different multi-level models from different domains to interconnect ontologically.

10 Conclusion

In the literature, the main claimed advantage of multi-level modeling over traditional modeling approaches is reducing "accidental complexity". Fred Brooks, to which the notion of accidental complexity is usually attributed, also identified the basic property that a well-designed language should possess to facilitate this goal – "conceptual integrity" (Brooks Jr 1995). According to Brooks, conceptual integrity is characterized by "simplicity and straightforwardness" and must ensure "unity of design" in which "every part must reflect the same philosophies and the same balancing of desiderata". Marc Lankhorst, one of the designers of the ArchiMate Enterprise Architecture Modeling language, elaborated on this concept by identifying the four basic design principles entailed by the notion of conceptual integrity (Proper et al. 2005):

- orthogonality - do not link what is independent,
- generality - do not introduce multiple functions that are slightly divergent,
- economy (a.k.a parsimony) - do not introduce what is irrelevant,
- propriety - do not restrict what is inherent.

The solutions to the Challenge presented in this paper demonstrates that, overall, the *LML /DOCL* modeling approach successfully applies these principles in a way that naturally leads to models with high conceptual integrity. First, the *OCA* architecture underpinning *LML* models directly applies the orthogonality design principles (in name as well as substance) for the express purpose of separating the independent notions of linguistic and ontological classification. Second, *LML*'s principle of combining different facets of a single underlying concept into a single model element (e. g., clabjects, connections, deep attributes) rather than separate model elements (e. g., classes/objects, associations/links, attributes/slots), directly applies the second design principle of generality by obviating the need to use multiple highly-specific model elements to represent a concept when fewer, more general model elements will suffice. Third, *LML*'s fundamental notion of level-agnosticness and the avoidance of unnecessary modeling constructs directly applies the design principle of economy. For example, the three vitality attributes can, between them, represent the same information as a large array of modeling concepts in UML (i. e., abstract class labels, tags, tagged values, static variables, etc.), while the basic ability to change and extend metamodels on-the-fly through inheritance avoids the need for all the UML modeling concepts associated with extensions (stereotypes, extension, etc.). Finally, by allowing multiple classification levels to be directly reflected in models, as well as the deep characterization control applied by general concepts over their more concrete offspring, *LML* directly applies the fourth design principle of propriety. This allows complex, deep characterizations scenarios in real-world domains to be represented without the usual array of patterns and workarounds needed when using traditional

two-level modeling approaches (De Lara et al. 2014b).

The efficacy of *LML*'s approach is evidenced by the way it can support complete solutions to the two Challenge scenarios that (a) basically provide the full look-and-feel of UML class diagrams using an extremely small underlying (meta)model (i. e., the linguistic metamodel shown in Fig. 3), and (b) are extremely concise and minimalist, both in terms of the total number of model elements used, and the number of different kinds of modeling constructs involved.

Nevertheless, as pointed out in Sect. 7, and elaborated in Sect. 8, the basic problem of reconciling strict modeling with the need to naturally represent different modeling spaces has yet to be addressed, at least in a pragmatic way, and is arguably the Achilles heal of *LML*'s version of deep modeling. This problem forced us to use an unnatural "trick" (i. e., workaround) to capture misaligned classification hierarchies in the ACME SE process example, which detracts from our solutions' claim to conceptual integrity. In this paper, we have therefore discussed various pragmatic ways of overcoming this misalignment problem in the context of our solution to the Challenge, and identified a deep, SUM-based modeling approach as potentially offering the most effective balance between the various "forces" at play. At this point it has to be mentioned that the non-strict SUM approach we presented as part of an alternative solution to misalignment problem does not yet have tool support and we have not yet set the rules that control how the locally strict views interact with a non-strict SUM. In our future work, we therefore plan to explore this approach further.

References

Almeida J. P. A., Kühne T., Rutle A., Wimmer M. (2019) The MULTI Process Challenge – EMISAJ Special Issue Version en. In: p. 4

Asikainen T., Männistö T. (Sept. 2009) Nivel: A metamodelling language with a formal semantics. In: Software and System Modeling 8, pp. 521–549

Atkinson C., Gerbig R. (2016) Flexible deep modeling with melanee. In: Modellierung 2016-Workshopband

Atkinson C., Kennel B., Goß B. (2011) Supporting Constructive and Exploratory Modes of Modeling in Multi-Level Ontologies. In: Procs. 7th Int. Workshop on Semantic Web Enabled Software Engineering)

Atkinson C., Kühne T. (2001a) Process and Products in a Multi-Level Metamodeling Architecture. In: International Journal of Software Engineering and Knowledge Engineering : SEKE ; IJSEKE 11(06), pp. 761–783

Atkinson C., Kühne T. (2001b) The essence of multilevel metamodeling. In: International Conference on the Unified Modeling Language. Springer, pp. 19–33

Atkinson C., Kühne T. (Oct. 2002) Rearchitecting the UML Infrastructure. In: ACM Trans. Model. Comput. Simul. 12(4), pp. 290–321

Balaban M., Khitron I., Maraee A. (2018) Context-aware factors in rearchitecting two-level models into multilevel models. In: Computer Software 1, p. 1

Brooks Jr F. P. (1995) The mythical man-month: essays on software engineering. Pearson Education

Carvalho V. A., Almeida J. P. A., Fonseca C. M., Guizzardi G. (2015) Extending the foundations of ontology-based conceptual modeling with a multi-level theory. In: International Conference on Conceptual Modeling. Springer, pp. 119–133

Clark T., Frank U. (2020) Multi-level Modelling with the FMMLx and the XModelerML. In: Modellierung 2020

Clark T., Neumayr B., Rutle A. (2018) 5th International Workshop on Multi-Level Modelling MULTI 2018 https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/

De Lara J., Guerra E. (2010) Deep meta-modelling with metadepth. In: International conference on modelling techniques and tools for computer performance evaluation. Springer, pp. 1–20

De Lara J., Guerra E., Cobos R., Moreno-Llorena J. (2014a) Extending deep meta-modelling for practical model-driven engineering. In: The Computer Journal 57(1), pp. 36–58

De Lara J., Guerra E., Cuadrado J. S. (2014b) When and how to use multilevel modelling. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 24(2), p. 12

Eclipse Foundation (2021) The Community for Open Innovation and Collaboration | The Eclipse Foundation en. https://www.eclipse.org/. Last Access: Last Access: 2021-05-07

Eker J., Janneck J. W., Lee E. A., Liu J., Liu X., Ludvig J., Neuendorffer S., Sachs S., Xiong Y. (2003) Taming heterogeneity-the Ptolemy approach. In: Proceedings of the IEEE 91(1), pp. 127–144

Eriksson O., Henderson-Sellers B., Ågerfalk P. J. (2013) Ontological and linguistic metamodelling revisited: A language use approach. In: Information and Software Technology 55(12), pp. 2099–2124

Fonseca C. M., Almeida J. P. A., Guizzardi G., Carvalho V. A. (2018) Multi-level conceptual modeling: From a formal theory to a well-founded language. In: International Conference on Conceptual Modeling. Springer, pp. 409–423

Frank U. (2014) Multilevel modeling. In: Business & Information Systems Engineering 6(6), pp. 319–337

Frank U. (2018) Toward a unified conception of multi-level modelling: advanced requirements.. In: MODELS Workshops, pp. 718–727

Gamma E., Helm R., Johnson R., Vlissides J. M. (1994) Design Patterns: Elements of Reusable Object-Oriented Software, 1st ed. Addison-Wesley

Gerbig R. (2017) Deep, seamless, multi-format, multi-notation definition and use of domain-specific languages. PhD thesis

Gonzalez-Perez C., Henderson-Sellers B. (2006) A powertype-based metamodelling framework. In: Software & Systems Modeling 5(1), pp. 72–90

Igamberdiev M., Grossmann G., Stumptner M. (2016) A Feature-based Categorization of Multi-Level Modeling Approaches and Tools.. In: MULTI@ MoDELS, pp. 45–55

Jarke M., Gallersdörfer R., Jeusfeld M., Staudt M., Eherer S. (1995) ConceptBase—a deductive object base for meta data management. In: Journal of Intelligent Information Systems 4(2), pp. 167–192

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: MODELS '20 : Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings

Jonkers H., Band I., Quartel D. (2012) The ArchiSurance case study. In: The Open Group, pp. 1–32

Kennel B. (2012) A Unified Framework for Multi-Level Modeling. PhD thesis

Kühne T. (2018) Exploring Potency. In: MODELS '18. ACM, Copenhagen, Denmark, pp. 2–12

Kühne T. (2022) Multi-dimensional multi-level modeling. In: Software and Systems Modeling, pp. 1–17

Kühne T., Lange A. (2020) Meaningful metrics for multi-level modelling. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–9

Lamo Y., Wang X., Mantz F., MacCaull W., Rutle A. (2012) DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment In: Computer and Information Science 2012 Lee R. (ed.) Springer, pp. 37–52

Lange A., Atkinson C. (2019) On the Rules for Inheritance in LML. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 113–118

Lankhorst M. M., Proper H. A., Jonkers H. (2009) The architecture of the archimate language. In: Enterprise, business-process and information systems modeling. Springer, pp. 367–380

Levendovszky T., Lengyel L., Mezei G., Charaf H. (2005) A systematic approach to metamodeling environments and model transformation systems in VMTS. In: Electronic Notes in Theoretical Computer Science 127(1), pp. 65–75

Macías F., Rutle A., Stolz V., Rodríguez-Echeverria R., Wolter U. (2018) An approach to flexible multilevel modelling. In: Enterprise Modelling and Information Systems Architectures (EMISAJ)–International Journal of Conceptual Modeling: Vol. 13, Nr. 10

Mylopoulos J., Borgida A., Jarke M., Koubarakis M. (1990) Telos: Representing knowledge about information systems. In: ACM Transactions on Information Systems (TOIS) 8(4), pp. 325–362

Neumayr B., Grün K., Schrefl M. (2009) Multi-level domain modeling with m-objects and m-relationships. In: Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling-Volume 96, pp. 107–116

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

Object Management Group (2021) About the Object Constraint Language Specification Version 2.4 https://www.omg.org/spec/OCL/2.4/ Last Access: 2021-05-03

Proper H., Hoppenbrouwers S., van Zanten G. V. (2005) Enterprise Architecture at Work: Modeling, Communication and Analysis, ed. M. Lankhorst

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: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 119–127

The Open Group (2021) TOGAF | The Open Group Website. https://www.opengroup.org/togaf. Last Access: Last Access: 2021-05-09

Theisz Z., Bácsi S., Mezei G., Somogyi F. A., Palatinszky D. (2020) Join potency: a way of combining separate multi-level models. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–5

Tunjic C. (2021) A Deep Orthographic Modelling Environment. PhD thesis

Varró D., Pataricza A. (2003) VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). In: Software & Systems Modeling 2(3), pp. 187–210

Volz B., Jablonski S. (2010) Towards an open meta modeling environment. In: Proceedings of the 10th workshop on domain-specific modeling, pp. 1–6