Multi-Level Design of Process-Oriented Enterprise Information Systems

Essential Guidelines and two Case Studies based on the FMML^{x} and the XModeler^{ML}

Ulrich Frank[*,a], Tony Clark[b]

[a] Universität Duisburg-Essen, Germany
[b] Aston University Birmingham, United Kingdom

Abstract. *This paper presents prototypical multi-level models of two uses cases. They comprise models of business processes and models that represent the context required to execute a business process. On the one hand, the context consists of the organizational units that are responsible for the execution of processes. They are represented by a model of organizational structures. On the other hand, the context includes the artifacts that are needed or manipulated by processes. The models serve to demonstrate the specific power of multi-level modeling. First, they integrate models on higher levels, which correspond to domain-specific modeling languages, with those on lower levels. Second, models are supplemented with objects on L0 to demonstrate how these can be integrated. Third, the models are executable without the need to generate code, since models and corresponding program code share the same representation, thus demonstrating the possibility of advanced application system architectures, which allow users to navigate a comprehensive representation of the system they work with at runtime. The presentation of the models is supplemented with a general evaluation of multi-level concepts. The design of the models was inspired by the EMISA process challenge. Therefore, they are evaluated against the requirements defined with the challenge. In addition, a challenge is discussed that goes beyond the challenge, that is, the design of multi-level models of behavior.*

Keywords. DSML • information system architecture • modeling productivity • executable models • models at runtime

Communicated by Peter Fettke. Received 2022-04-08. Accepted on 2022-05-03.

1 Introduction

Research on multi-level modeling has produced a considerable number of relevant contributions. This applies to specific languages and tools (Atkinson and Kühne 2008; Frank 2014b; Jarke et al. 1995; Kühne and Schreiber 2007; Lara and Guerra 2010; Macías et al. 2018; Neumayr et al. 2009, 2018; Volz 2011) as well as to their application (Frank 2016; Igamberdiev et al. 2016; Jeusfeld 2019; Kaczmarek-Heß and Kinderen 2017; Kinderen and Kaczmarek-Heß 2020; Lara et al. 2014; Rossini et al. 2015; Selway et al. 2017). Comparing the approaches is sometimes difficult because of their complexity and the fact that terminology is not yet unified. However, in order to achieve a comprehensive assessment of the current state of research, a differentiated comparison of the approaches combined with an analysis of essential commonalities is of central importance. At the same time, exemplary applications are needed to illustrate the potential of multi-level modeling over traditional approaches. The MULTI chal-

* Corresponding author.
E-mail. ulrich.frank@uni-due.de

lenges that started in 2017[1] provide an excellent opportunity to support both objectives (cf., e. g., (Jeusfeld 2019; Mezei et al. 2018; Rodriguez and Macias 2019; Somogyi et al. 2019)).

This paper was prepared as a contribution to the EMISA process challenge, which is an updated version of an earlier MULTI process challenge (Almeida et al. 2019). The organization of this contribution follows the guidelines published with the process challenge. However, the solutions presented in this paper go well beyond the requirements specified with the challenge in parts, which makes a direct comparison with other solutions difficult. Therefore, the editors have recommended that this contribution should not be included in the challenge special issue in this form. For this reason, it appears in a regular issue. Nevertheless, three measures were taken to foster the comparison with other solutions. First, a thorough analysis of the requirements defined with the challenge is presented to show why a few details of the requirements haven been adapted. Second, the paper includes a comprehensive assessment of the presented models against the requirements defined with the challenge. Finally, we give an overview of those aspects of the presented solution that we decided should go beyond the challenge in order to demonstrate the potential of multi-level modeling more clearly. In addition, the paper includes a supplemental section where we present preliminary ideas on how to design multiple levels of process models. We account for all aspects of the requirements defined for the challenge.

The following guidelines identify key aspects for those readers who are not interested in specific details of the solution and, therefore, might be offended by the extent of the paper. The details of the language architecture and the tools presented in Sect. 2 can be omitted by readers who are primarily interested in the presented solution. This is the case, too, for the code fragments and constraints shown in the paper. It should be sufficient for those readers to look at the overview of language concepts and the corresponding notation shown in Fig. 2, which is also suited as a reference when studying the models presented in the paper. The discussion of selected requirements in Sect. 3 is relevant for understanding specific aspects of the proposed models. For readers who aim at developing a more general understanding it should be sufficient to refer to the requirements presented with the challenge. We recommend reading the design guidelines presented in Sect. 4.1, because they provide the foundation for most of our design decisions. Fig. 5 gives an overview of the architecture of the presented multi-level model. It serves as an additional guideline for selective reading. Readers who are interested in the software development case only may omit those figures and related sections (4.2.3, 4.3.2, 4.4.4) that focus on the insurance case – et vice versa. The evaluation of the presented solution includes a comparison with traditional language architectures that can be omitted by readers who are familiar with multi-level modeling. Finally, in Sect. 6 we discuss a key aspect of process modeling, namely the creation of multi-level models of behaviour. We would hope that it is especially relevant for those readers who are interested in enriching process modeling languages with abstraction.

2 Background: Language, Tool and Terminology

Our contribution to the challenge is designed with the FMML^{x} and implemented with the language engineering and execution tool XModeler^{ML}. Both have been described in various publications already. Therefore we will give a brief overview only. We will also describe core concepts of the terminology we use, since there is still lack of a unified terminology for multi-level modeling. Also, an overview of the terminology should support those readers that are not familiar with multi-level modeling to better understand the presented models.

2.1 Notes on terminology

Every class in a multi-level model is an object at the same time since it has state and can execute

[1] https://www.wi-inf.uni-duisburg-essen.de/MULTI2017/#challenge, accessed on 2022-02-20

operations. Creating a class from its metaclass corresponds to instantiation, but is not the same, because not all of its properties are instantiated directly. Instead, the instantiation of some features, which we refer to as *intrinsic*, may be deferred. Therefore, we follow a suggestion made by Neumayr et al. (Neumayr et al. 2009; Neumayr and Schrefl 2009) and speak of **`concretization`**. Accordingly, the act of classifying objects is different from traditional object-oriented language architectures like the one advocated by the MOF. To express this difference, we speak of **`intrinsic classification`** in the case of multi-level models. Intrinsic features in the FMML^x comprise attributes, operations, associations and constraints. Associations may be defined between classes at different levels.

While the term "level" is of essential relevance for multi-level modeling, it still lacks a unified definition (Kühne 2018). We use a pragmatic approach here and regard a level as being comprised of classes that allow the same number of concretization steps until the lowest level is reached. To distinguish this concept of level from the levels proposed by the MOF, we use the denominator "L" instead of "M", e. g. L2 instead of M2. There are different approaches in use to specify levels. Some approaches assign a level directly to a class (Atkinson and Kühne 2001; Frank 2014a). While levels are usually expressed by numbers (either starting at the bottom or at the top), they may also be represented by terms that are supposed to clearly symbolize the intended level of classification (Neumayr et al. 2009). We use integers, starting with 0 at the lowest level. Other approaches favor an implicit definition of levels, cf. (Balaban et al. 2018; Jácome-Guerrero and Lara 2020). All these approaches have in common that a level corresponds to a certain value of an explicit or implicit ordinal scale.

We refer to the tree of classes that are concretized from a class and its concretizations as a "concretization subtree". We call an object O that is concretized (or instantiated) from a class which in turn is concretized from a further class and so on to a class A a *descendant* of A. Accordingly, we call A an *ancestor* of O.

The upcoming new version of the FMML^x also allows for contingent level classes, that is, classes the level of which may depend on their usage context. Similar approaches to increase the flexibility of multi-level models are presented in (Neumayr et al. 2018) and (Guerra and Lara 2018). However, we did not need contingent level classes for the models presented in this paper. With respect to processes we use the terms "activity" and "task" synonymously.

2.2 Language and Tool

The models presented in this paper were created with the Flexible Multi-Level Modeling and Execution Language (FMML^x), (Frank 2014b). It is specified through an extension of XCore, the meta model of the multi-level language engineering environment XModeler (Clark et al. 2015a, p. 40). As a consequence, the FMML^x is implemented in the XModeler, which resulted in a new version of the tool, the XModeler^{ML} [2] . In addition to using a multi-level modeling environment, the contribution also builds on previous work on DSMLs for enterprise modeling, in particular on the meta-model that specifies a language for business process modeling, the MEMO OrgML (Frank 2011b). The metamodel that defines the abstract syntax and semantics of the MEMO OrgMLwas originally specified with the MEMO meta-modeling language (Frank 2011c), which is not a multi-level language. The model of the relevant organizational context is, in part, a multi-level reconstruction of the MEMO OrgML for modeling organizational structures (Frank 2011a).

The XModeler is a language engineering workbench whose core language is both reflexive and extensible. In this way, the XModeler is both an instance of itself and a basis for defining a wide

[2] The XModeler^{ML} as well as various screencasts that demonstrate the use of the tool are available on the web pages of the project LE4MM (Language Engineering for Multi-Level Modeling): https://www.wi-inf.uni-duisburg-essen.de/LE4MM/

range of co-existing language variants. FMML^x is an example of such a variant.

The XModeler core is an object-oriented "super-language" (Clark et al. 2015b) called XOCL whose class-based type system is called XCore that supports the creation of classes on any level. Classes are also objects and the level of a class is computed dynamically on demand depending on its use. The design of the FMML^x was based on the assumption that in most cases the level of a class is relevant for its proper interpretation. Therefore, a class created with the FMML^x has a particular level that cannot be changed (except for classes that are explicitly defined as level-contingent). Like other multi-level modeling languages, the FMML^x allows for an arbitrary number of levels.

FMML^x is implemented as an extension of XCore as shown in Fig. 1. Attributes `isIntrinsic` and `instLevel` are added to XCore classes in order to represent features (attributes, operations), whose instantiation is deferred to lower levels. For intrinsic associations the instantiation levels can be defined separately for both participating classes (specified with instances of `End`). The attribute `isCore` serves a specific model management purpose that is of no further relevance.

To enable explicit levels, there is need for specific instantiation operations and for storing a level with every object. This is achieved by extending XCore with two specific classes. `MetaAdaptor`, which is instantiated from the XCore class `Class`, and inherited from it as well, serves the definition of two specific instantiation operations. The operation `new()` overrides `new()` in `Class`. It concretizes an object from a class on level n and assigns it the level n-1. In addition, the operation `newAtLevel(l: Integer)` allows the creation of an object on a specific level. The `MetaClass` serves the execution of the instantiation operations. It also includes the attribute `level` that is inherited to all objects in is concretization subtree.

In addition, `MetaClass` includes various generic operations, which in part override corresponding operations in `Class`, such as `allInstances()`. Note that the attribute `lastUpdated` was added especially for the challenge (see requirement P19 in Tab. 1). It is inherited to all classes in the concretization subtree of `MetaClass`. Therefore, a corresponding slot is available in every object that is part of an FMML^x model. Note that the name of the attribute is not shown with the representation of a class in a diagram. That corresponds to other generic properties like "name" or "allInstances()". The corresponding slot value can be shown in a diagram upon user request. In general, this monotonic extension mechanism allows the definition of new properties that apply to all classes without the risk of side-effects.

The FMML^x also features delegation as a specific kind of association. Within a delegation association, one class serves as delegatee class, the other as delegator class. Every message an instance of the delegator class receives that does not correspond to any of the operations offered by it is forwarded transparently to instance of the delegatee class it is linked to.Through its implementation with XCore, the FMML^x is executable. Unlike traditional model editors such as UML editors, the objects that implement a model in the editor are not located on M0, but on the level they represent conceptually, thereby allowing for a common representation of program and model, where code, diagrams, and further representations are different views on the same system. As a consequence, there is no need to generate code and to bother with synchronisation of model and code.

FMML^x levels are distinguished by different shades of blue. The darker the shade of blue shown as the background of a class name the higher the level it indicates. In addition, the level is indicated through the respective number printed in gray. The diagram in Fig. 2 serves the description of the notation and the illustration of essential language concepts. It also demonstrates that multi-level modeling editors overcome traditional boundaries between modeling language and model. Every class that is created with the model editor is a concept of the multi-level language at the same time. Therefore the palette, which is fixed for traditional languages, is dynamically adapted during the creation of a model. That

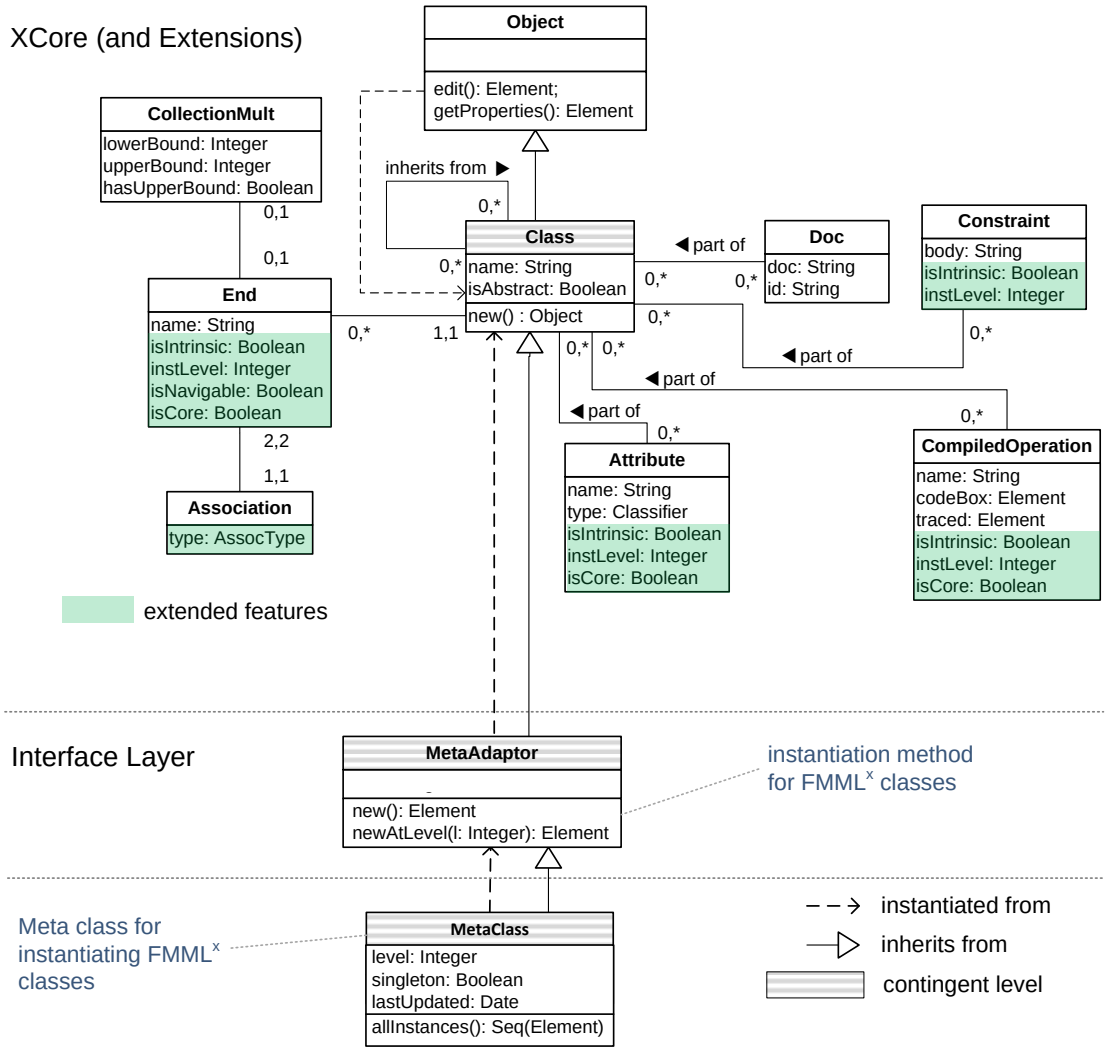

Figure 1: Meta model of the FMML^x and its relation to XCore

corresponds to the natural process of developing a common technical language in a project. Such a process will usually build on generally accepted concepts as they are, e. g., defined in textbooks. Those are usually refined to more specific concepts that fit specific aspects of the domain. Eventually, the concepts are used to classify particular objects, and to refer to those. The latter aspect is also accounted for in the FMML^x . A model may include objects on L0. Objects on L0 will usually not qualify as a conceptual abstraction. Nevertheless, it can be very useful to integrate them with the conceptual model they are based on. Such an integration does not only promote reflection of objects on L0 (and above), that is, asking them for their class, metaclasses, etc., it also opens the possibility to change those classes during runtime, which, of course, recommends a very cautious approach especially to deleting parts of a model.

The example model in Fig. 2 also shows that the classes of a model may be defined on different levels. While classes like **`Person`** or **`Employee`** are located on L1, the class **`Product`**, also instantiated from **`MetaClass`**, is on level 3. This corresponds directly to the structure of natural languages or,

more precisely, of the use of language in domains of discourse. Some concepts, like person, represent common, rather general knowledge that is sufficient for a given purpose. Therefore there will often be no need to refer to more general terms (like, e. g., animate being) or to account for more specific ones (like, e. g., teenager). At the same time, more elaborate discourses may require distinguishing between terms on different levels of abstraction. If, e. g., product managers talk about products, they need to distinguish more general uses of the term from more specific ones.

Since every class in the model is an object, it may have a state. The values of the corresponding slots, like **amount** in **i1**, can be shown in the diagram if this option is selected. The values produced by operations can be shown in the diagram, too, e. g., **invoiceTotal()**. The diagram gets updated as soon as the model changes. A class can be assigned constraints, the evaluation of which can be deferred to a specific level. If a constraint is violated, a constraint report is presented with the affected object. Note that the example is by no means related to the challenge. It serves illustration purposes only.

Every model with classes above L0 can be concretized (or instantiated respectively) within the XModeler^{ML} . That implies that objects which are concretized/instantiated from two classes that are associated need to be linked. Otherwise the corresponding model could not be executed. Therefore, associations need to be defined as unidirectional or as bidirectional, which is indicated by the direction of the edge that represents an association. Depending on this definition, uni- or bidirectional references are created. Furthermore, associations can be defined as intrinsic. Each of the two classes involved in an intrinsic association can be assigned a specific instantiation level. For example, the association **referstTo** between **InvoiceItem** on L1 and **Product** on L3 is to be instantiated on L0, which means that on L0 concretizations of **InvoiceItem** are linked to descendants of **Product**. Note that it is also possible to associate/link classes at different levels. Links represent instantiations of associations, that is, they link objects that were concretized from the classes that participate in an association. The designator of a link as well as the navigability correspond to the definition of the association it was instantiated from. The diagram editor is equipped with filters that allow to define what parts of a model are shown in a diagram.

As soon as an attribute or association is specified, the XModeler^{ML} generates access operations. Operations can be specified and compiled either within the model browser or within specific editors available with the diagram editor. The following example shows the XOCL code to implement the operation **invoiceTotal()** of the class **Invoice**. To support users with a clear arrangement of diagrams, the tool allows to selectively hide certain model properties such as, e. g., methods, access methods or inherited properties. Note that in all diagrams shown in this paper, generated access methods are faded out.

The following example serves to illustrate the implementation of operations within FMML^{x} models:

```
context Invoice
  @Operation invoiceTotal[monitor=true]():
    Float
    let sum = 0
    in @For i in self.getInvoiceItem() do
         sum := sum + i.itemTotal())
       end;
     sum
  end
end
```

Since not every execution of an operation requires updating an object diagram, monitors can be defined that update the diagram upon the execution of certain operations. This is done by including "[monitor=true]" in an operation's definition. The effect is shown in Fig. 2, e. g., with the visualization of the value computed by **invoiceTotal()** of the object **invoice1**.

The XOCL also serves the specification of constraints. A constraint is defined as a boolean expression and a corresponding constraint report that is presented in case a constraint fails. The following example illustrates the specification of constraints:

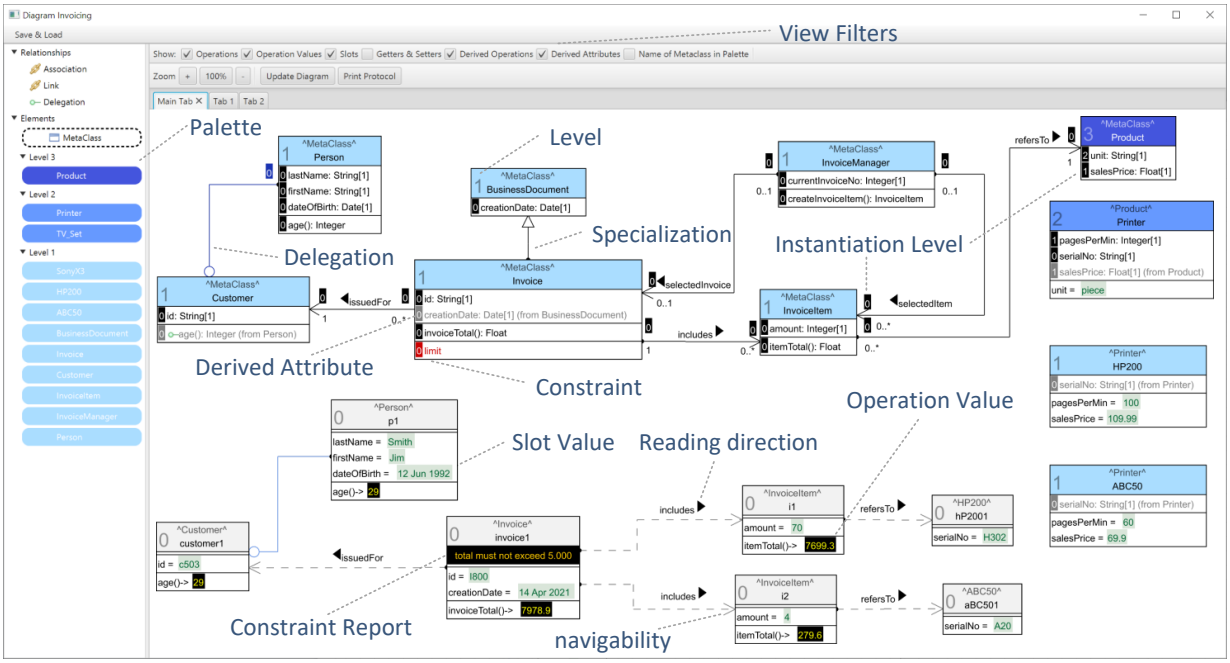

Figure 2: Example object model created with the FMML^x

```
context Invoice
  @Constraint limit
    self.invoiceTotal() <= 5000
    reason: "Total amount of invoice must
    not exceed 5000."
  end
```

The XModeler^{ML} gives its users the choice between different kinds of model representations. As an alternative to the diagram editor, or in addition to it, various browsers are provided that allow navigating and changing a model. Fig. 3 shows an instance of the model browser that is opened on the same model as the diagram in Fig. 2.

In addition to the data types that are provided by the XModeler^{ML} , the FMML^x is supplemented with an extensible set of auxiliary types such as monetary values or currencies and enumeration types that comprise of an ordered collection of values to characterize specific properties. Advanced users may also use a console that allows to inspect and manipulate objects using the XOCL. References to objects in a model can either be established through global variables or by specifying the path from the system's root to the targeted model.

3 Case Analysis

This section serves a brief analysis of the cases presented in the challenge. Note that we shall comment on those aspects only that leave room for interpretation, not on all requirements described in the challenge:

P4: The term "actor" represents a versatile abstraction that may relate to persons or machines. P4 indicates that it is intended to represent people. However, P5 recommends a slightly different interpretation: an actor represents the holder of a position or an organizational role. In most cases, an actor will be an employee. We decided for this interpretation for two reasons. First, it corresponds more clearly to the technical terminology of organizational design. Second, it enables a more elaborate definition of organizational assignments, since it distinguishes between permanent organizational units (positions) and temporary responsibilities (roles).

P5: corresponds to the comment on P4.

P6: Accordingly, this requirement would be translated into: "A task type may alternatively be

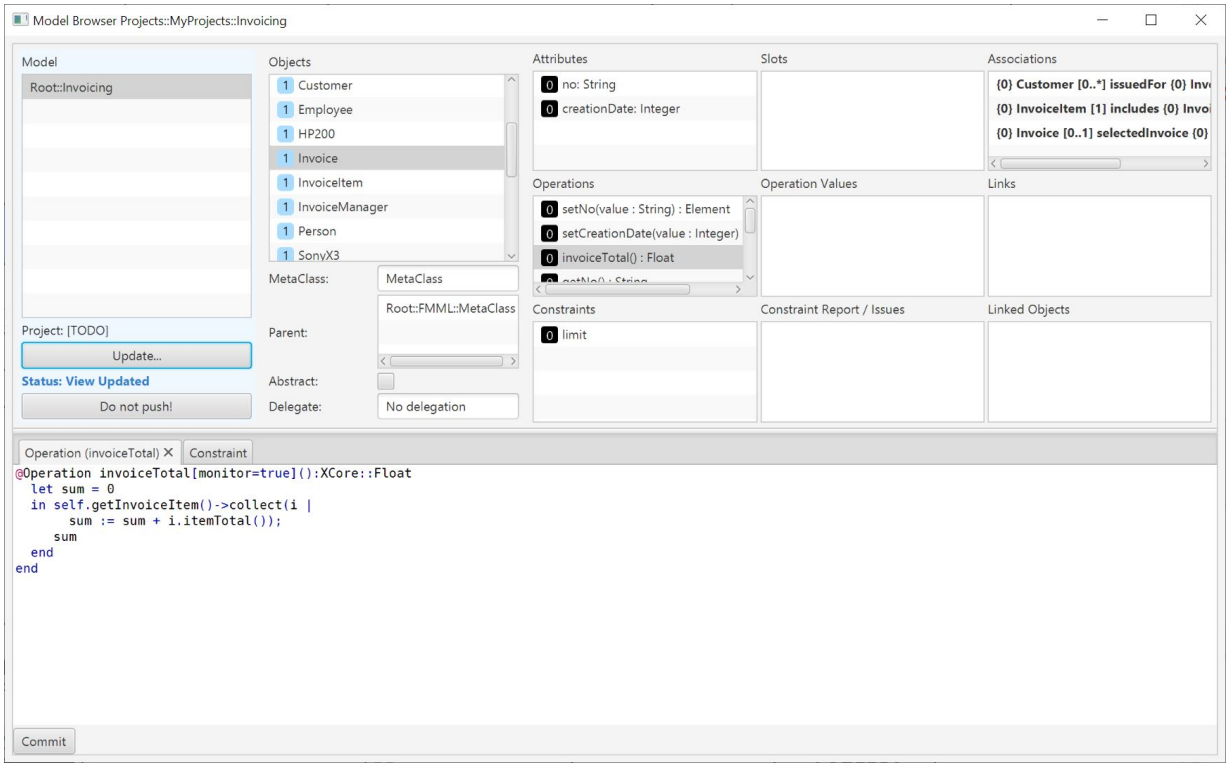

Figure 3: Illustration of the XModeler^{ML} model browser

assigned to a particular set of positions that are authorized ...".

P7: While a decision can be regarded as an artifact, we prefer conceptualizing it as a set of alternative events with additional information regarding the type of a decision.

P9: While seniority is a common concept, it is not obvious how to interpret it in this context. One could think of the total time a person was employed or the time she served in a particular position or role. Both options as well as a combination of both can make sense. However, we assume that seniority is captured through specific position types such as "senior manager", "project leader", or "financial officer" (which are mentioned in the challenge) and, possibly, additional ones, e. g., "senior developer".

P14: The requirement to instantiate each artifact type in *every* activity (task) seems too restrictive. If a customer order is passed from one process activity to another this requirement would lead to useless instantiations. However, we assume that it can be useful to define for specific task types that their instances are supposed to create instances of specific artifact types.

P15: We interpret this requirement as follows: an actor may hold a position and additional roles.

P16: In a strict sense the technology we use does not satisfy this requirement because every object (or class) is of one class only. The language we use does not support multiple generalization.

P18: While this requirement makes sense in many cases we would be reluctant applying it in general. There may be cases where a lower level manager is authorized to perform tasks that require a specific technical competence such as deploying a new version of a database schema. It is not necessarily a good idea to grant a senior manager the authorization to perform any task a lower level manager is authorized to support. In addition there is no connection between instances of a subclass and instances of a corresponding superclass. If

the class **Senior Manager** is specialized from the class **Manager**, and **Manager** is associated with a class **Product** to express that instances of **Manager** are linked to those products (or product types, if **Product** is on a classification level > 1), for which they may define prices, then an instance of the **Senior Manager** would have no link to that instance of **Manager**. Therefore, it would not be clear, what authorization to "inherit". But even without associations, specialization would compromise system integrity. If, e. g., an upper limit for salaries of managers is defined with Manager, the corresponding assertion: "Every manager has a maximum salary of x" would need to hold for every senior manager, too, but very likely should not. As a consequence we handle this requirement with care.

S3: The task **Coding** should have a reference to code. Since code must reference the programming language in which it was written (S4), we assume that no further, redundant references from **Coding** to programming languages are required.

S7: This might reflect a rule that is valid in a certain time frame but it hardly qualifies as a constraint that should be built into software because that would require a program modification when Ann Smith leaves the company or somebody else is authorized to write COBOL code. Our approach to satisfy this rule is to reformulate it: Only developers who master COBOL are allowed to change COBOL code. That would imply that Ann Smith is currently the only developer that masters COBOL.

S11: Since Bob Brown is an analyst we assume that analysts in general may design tasks.

S13: We follow the guideline and assume that S13 overrides S2.

It is not defined which role or organizational unit performs the design activity. We assume that holders of the position "Designer" are responsible for that activity.

4 Model Design

The design of the proposed solution is separated into two parts. The first part provides the multi-level context for the process models including the organizational structure and artifacts that are consumed and produced by processes. The second part provides models for the software engineering process and the claims handling process.

4.1 General Orientation and Essential Design Guidelines

The design of multi-level models integrates what is traditionally regarded as conceptual modeling with the design of modeling languages. In other words, there is no strict dichotomy between language and language application. While such an approach may seem odd to those who take the traditional approach for granted, it is actually an obvious reflection of how technical languages are organized in advanced societies. As shown in Fig. 4, a field of expertise is typically characterized by a general terminology as it is described in text books that give an overview of the entire field. While concepts on this level are useful, that is, can be reused in a wide range of specific cases, they will often not be elaborate enough to satisfy specific needs. They require *refinement* for more specific technical languages. This kind of refinement can be repeated step by step until a level of specificity is reached that fits a particular use case. Such a language hierarchy combines the benefits of a wider range of reuse on the higher levels with a higher degree of productivity on the lower levels where lower levels benefit from reusing concepts on higher levels. This idea of language levels serves us as an orientation for the design of multi-level models. Therefore, we introduce various concepts on higher levels that are not explicitly mentioned in the challenge. It is, of course, possible to represent a concept like manager as a class on M1 that is instantiated from a generic metaclass on M2. However, that would require defining it from scratch. Instead, we suggest defining it by re-using more general, but still domain-specific concepts like position, management position, process manager, etc., to constribute to modeling productivity (for those you need to specify a certain kind of management position) and model integrity. Different from a generic metaclass, a domain-specific metaclass

guides with the specification of more specific classes and prevents nonsensical specifications to a large degree. Note that the example in Fig. 4 relates to concepts relevant for the challenge, it does not anticipate the solution that we shall present later.

The design of a multi-level model is in general more demanding than the design of a traditional one level model. At the same time, there is lack of a comprensive method for designing multi-level models. De Lara and Guerra present design patterns for guiding the creation of multi-level models with regard to specific problems (Lara et al. 2014). Kühne proposes "sanity checks" that are based on a more formal analysis of models with specific emphasis on guiding a proper conception of level (Kühne 2018). According to our experience, it requires multiple loops to refine classes and their levels. Apart from that we use a few guidelines to support the required design decisions. They correspond to those presented with the design method in (Frank 2021). The following two guidelines are of specific relevance. They are adapted from more general guidelines that apply to the construction of any conceptual model:

Design principle 1: Specify known knowledge on the highest possible level within the scope of your project. *Rationale*: The representation of knowledge on level l that could be represented on a level m with m > l creates the risk of conceptual redundancy, which in turn compromises integrity and adaptability. It is important, though, to determine the highest possible level within the scope of a particular project, instead of aiming at the highest level in general.

Design principle 2: The higher the level of a class, the more invariant it should be. *Rationale*: In general, it is advisable to design classes on any level in a way that they are widely invariant during the life time of a system. However, the future of a domain may be hard to predict. The higher the level of a class, the more other classes are affected by its modification. Thoroughly check all properties of a class C in case they apply to possible future classes on levels below that of C.

These guidelines are specific to the design of multi-level modeling:

Design principle 3: The design of a multi-level model recommends combining a top-down with a bottom-up approach. *Rationale*: Our experience with the design of multi-level models has shown that modelers have different preferences and abilities regarding the concepts they start with. When thinking about the targeted domain, some associate higher level concepts at first, while other look for more concrete examples. In both cases, it is important to develop a hierarchy of concepts, which, at first, is created by using "is a" relationships only. During the course of model design, it is usually required to combine both approaches to iteratively assess and revise a given state of a model. Finally, "is a" relationships have to be disambiguated, which will eventually result in the construction of levels. Note that most modeling tools, including the XModeler^{ML} , do not directly support a bottom-up approach, since they require the class of an object to exist before it can be created.

Design principle 4: "Fake" levels should be avoided. If a class does not include any property that is instantiated on the level below, it should be modeled as a superclass. *Rationale*: There is clear semantic difference between generalization and intrinsic classification. That distinction would get blurred if clear cases of generalization are modelled as intrinsic classification. Note that this guideline will not exclude modeling a class as an intrinsic metaclass instead as a superclass if only foreseeable future changes may lead to properties that are to be directly instantiated. For example, it may seem appropriate at first to abstract the two classes `Printer` and `Scanner` into the common superclass `PeripheralDevice`. However, if we know that in the near future each product category (like printer and scanner) will be assigned a specific international category id, that would justify modeling `PeripheralDevice` as intrinsic metaclass of the two other classes.

The specification of business process models requires accounting for the relevant context. Processes make use of artifacts, which they may use

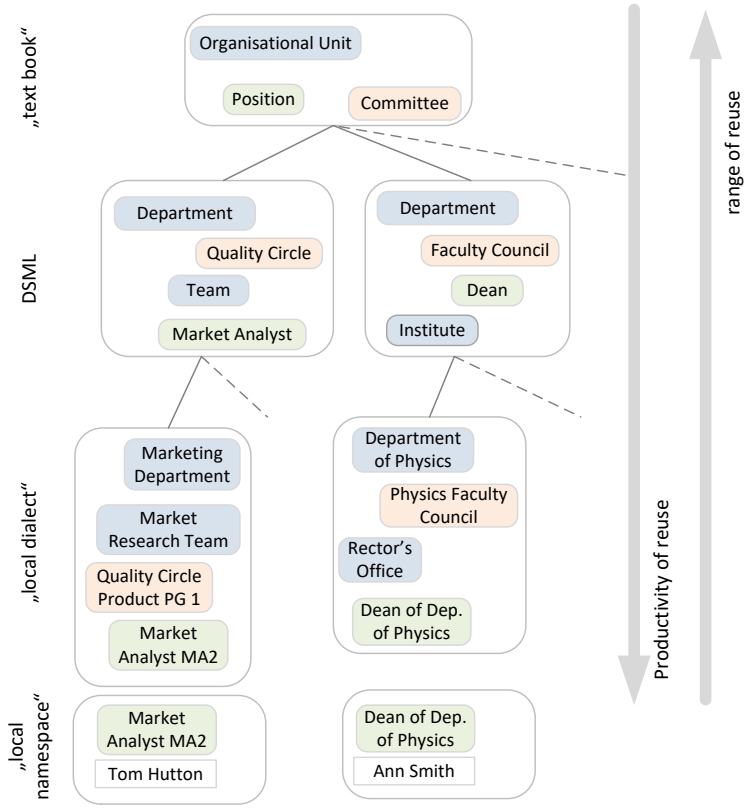

Figure 4: Illustration of levels in natural language

or create/manipulate. Furthermore, business processes will usually require the participation of human actors within an organizational setting. Accordingly, there is need to model organizational structures and artifacts in addition to the two process models. The design of the multi-level model presented in this paper reflects the idea shown in Fig. 4. Therefore, each domain, process, organizational structure, and artifact, is represented in a multi-level model. In each case, the highest level represents text book level knowledge, which is, step by step refined into more specific models. Fig. 5 gives an overview of the model components that form the overall contribution to the challenge. Since the process models refer to the context models, we shall start with the presentation of the context models. The presentation of all models follows a common structure. At first, we introduce core concepts. Subsequently, we discuss those design decisions that seem to be of specific relevance. Finally, if that is the case, we will discuss further issues such as specific lessons learned.

4.2 Organizational Context

The models of organizational structures we propose go in part beyond the requirements of the challenge. This is for two reasons. First, we regard it as an implicit request related to the design of multi-level models in general to account for reuse. That recommends developing abstractions that are suited to cover further possible use cases. Second, the meaning of an organizational unit depends on its relation to other organizational units. That may require modeling organizational units that are not explicitly mentioned in the challenge.

4.2.1 General Concepts

Core Concepts: An organizational structure consists of organizational units and relationships between these units. Relationships comprise ag-

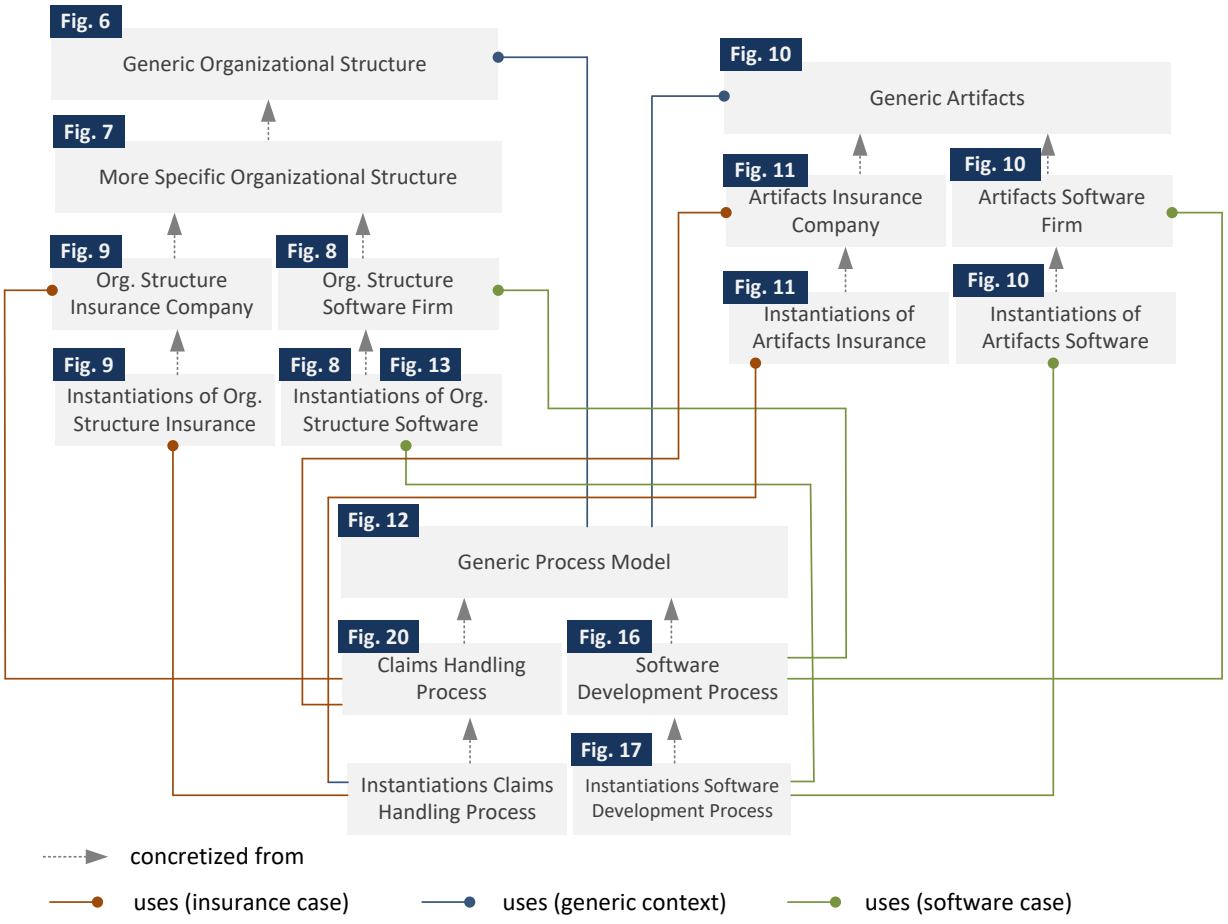

Figure 5: Structure of the overall multi-level model

gregation and, possibly, different kinds of line of command (such as disciplinary or related to a certain function or product). In addition to organizational units, roles and committees are important concepts to define the division of labour and its coordination within an organization. Since there is no need to cover committees in the scope of the challenge, we will not account for them. Organizational units exist either as composite units that include further organizational units or as elementary units, that is, as positions. The corresponding concepts are represented with the composite pattern on L3, which is, of course, not a "natural" choice, but a result of analyzing the variety of more specific concepts. Different from a position, a role is not regarded as an organizational unit in a strict sense, because it is not permanently assigned to an employee. A role may be filled only by holders of positions the types of which qualify for the corresponding role type. We also know that a particular position, represented by an object on L0, is filled by an employee and that an employee is a person. Some positions are characterized as management position, which means that holders of these positions may supervise holders of other positions. Projects are also often regarded as a specific kind of composed organizational unit, that is, however, different from regular organizational units, only of temporary nature. This textbook knowledge about organizations is shown in Fig. 6.

Concretizations of the classes shown in Fig. 6 still represent textbook knowledge, but on a more specific level. Concepts such as "department", "head department", or "division" are probably

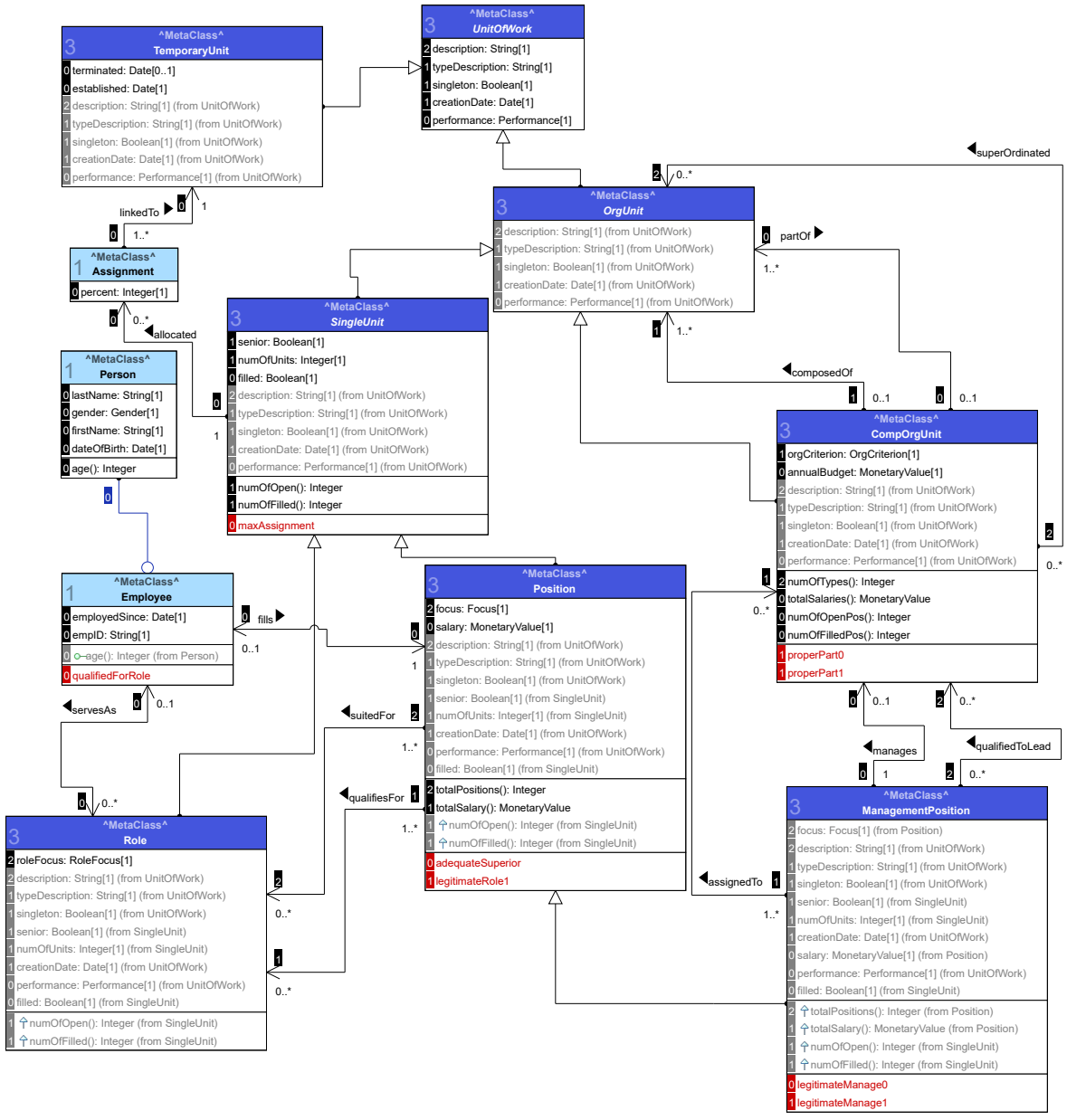

Figure 6: General concepts for modeling organizational structures

common in many countries. Apart from specific peculiarities, which may vary with the cultural background, we assume that the general use of these concepts comprises a common understanding of an order of aggregation. A subsection is part of a department, which may be part of a head department, etc. This knowledge is represented on L2 and forms a language layer that can be used to describe more specific types of organizational units, e. g. with respect to a certain industry or a particular company. That requires to clearly distinguish between these more specific types. The attribute **roleFocus** of the class **Role** and **focus** of the class **Position** serve this purpose. The intrinsic attribute **orgCriterion** of **CompOrgUnit** serves the same purpose, but for distinguishing organizational units on level 1 only. These attributes are to be instantiated on level 1.

Note that the idea of textbook knowledge does not imply that all classes representing this knowledge need to be on the same level. There may be cases, where the diversity of a concept is not wide enough for multiple levels of classification. In the example, this is the case for the classes **Person** and **Employee**. One could think of classifying people into men and women, which could be achieved by adding a further level. However, in the context of organizational design that will usually not be required. In the case of employees, one could think of multiple classification, e. g. with respect to qualification, authorization, responsibility, etc. However, these aspects are covered by the classes **Position** and **Role** and their multiple concretizations. While an employee is assigned to one and only one position, she may be assigned more than one role at a time.

In addition to positions, the model accounts for roles. A role type such as "Tester" serves describing a temporary assignment. It is possible to define which position types qualify for which role type. As a consequence, only those holders of a position the type of which qualifies for a certain role type can fill a corresponding role. While an employee must not hold more than one position, she may hold more than one role simultaneously. Note that it is conceivable to represent "Tester" as a position, too. However, according to the challenge it should be possible that an employee is both analyst (i. e., holds a corresponding position) and a tester. Since we decided to model "Analyst" as position, the only option left for "Tester" is role. The relationships between positions and roles are specified through associations that apply to different levels. First, the regular association **suitedFor** serves assigning position types on L2 to the role types on L2 they are suited for. The intrinsic association **qualifiesFor** between **Position** and **Role** allows expressing that a certain role type on L1 can be filled only with an employee who holds a position of certain types.

The class **ManagementPosition** is specialized from **Position**. The association **qualifiesToLead** with the **CompOrgUnit** allows defining the compositional unit types on L2 that a management position type on L2 is eligble to manage. Its concretization enables, e. g., expressing that a department manager qualifies as manager of a department — and, may be, in addition, of a project. The intrinsic association **assignedTo** serves linking particular management position types, such as **SoftwareProjectManager** to corresponding organizational unit types like **SoftwareProject**. Finally, the intrinsic association **manages** allows assigning particular management positions to particular organizational units. Hence, this conceptualization allows defining elaborate governance structures.

The excerpt of the model shown in Fig. 7 represents a partial concretization of the more generic model in Fig. 6. The concepts represented by classes on L2 should correspond to concepts used in many organizations, hence, they should allow for a wide range of reuse.

The multi-level model of concepts in Fig. 6 and 7 serves as a DSML to create more specific models of organizational structures.

Design Decisions: Filling organizational roles will usually require a certain qualification. Therefore, it may seem appropriate to associate the class **Role** with **Employee** to express whether or not an employee is qualified. We decided for a different conceptualization, where the qualification is defined by the association **suitedFor** and the intrinsic

Figure 7: General concepts for modeling more specific organizational structures

association **qualifiesFor**, both between **Position** and **Role**. This is mainly, because it allows for a higher level of expressiveness and abstraction. The association **suitedFor** allows defining that a certain kind of role, such as a role related to quality management (**QM_Role** in Fig. 7) requires somebody who holds a position that qualifies as "technical". The association **qualifiesFor** enables to further constrain this rule. Both would not be possible by associating **Role** with **Employee** on L1. But why did we then decide to assign an employee object to a role on L0 (via the intrinsic association **servesAs**)? We assume that this conceptualization is better suited to fit common ideas of filling roles.

Modeling composed organizational units leads to the question whether specific units like, e. g. "marketing department" should be modeled as instances on L0 or as types on L1. At first, it may seem strange to model a marketing department as a type, because there seems to be no need to further instantiate it. However, there are two reasons, why we decided for modeling those composed units as types. First, large companies that comprise various subsidiaries may want to define a certain organizational schema that is instantiated with every subsidiary. Second, distinguishing between marketing department and, e. g., manufacturing department on the type level allows for the specification of characteristic properties, e. g. that a marketing department must include a market research group. If a company has one organizational unit only, the corresponding classes would have to be defined as singletons.

Intrinsic associations require a more elaborate analysis. First, the regular association **superOrdinated** serves the definition of a hierarchy of composed organizational unit types. For example, the type "head department", represented by a

class on level 2, is characterized as being superordinated to the type "department", which in turn may be superordinated to the type "subsection". The intrinsic association **composedOf** is instantiated on level 1, which is to say that a corresponding link between two objects on level 1 is created. Hence, a class like **MarketingDepartment** could be linked to a **MarketResearch** to express that in a particular concretization, one could also speak of a specific domain, the organizational unit type "marketing department" comprises a sub unit type "market research". Note that for each of the corresponding classes it is possible to specify whether they are singletons or not (defined with the attribute **singleton** within the class **UnitOfWork**, see Fig. 6). Finally, the intrinsic association **partOf** allows linking objects that represent organizational units on level 0 to be linked. For example, a particular position can be assigned to a certain organizational unit, or, a particular subsection can be assigned to a particular department. Please note that in rare cases, the diagram editor places diagram elements on top of each other. That may lead to individual diagram elements being hidden, such as the multiplicity of the association **superordinated** between **OrgUnit** and **CompOrgUnit** in Fig. 6.

The introduction of the class **TemporaryUnit**, which can, e. g., be concretized into a class like **Project** represents the outcome of a difficult design decision. Projects are organizational units that are usually outside of the permanent organizational structure of a company and do not include other permanent organizational units. Instead, particular positions are assigned to projects for a limited time. That is the rationale for conceptualization we finally decided for. However, it is conceivable that projects have a different meaning in project-oriented organizations such as consulting firms. Therefore, the generalizability of the proposed conceptualization may be limited.

We assume that position types can be assigned to any composite organizational unit. Like any organizational unit type, a position type can be defined as singleton. For reasons outlined in 3, we do not represent the concept "actor" directly, but regard an actor as being represented by a certain position or a role each of which is filled by an employee. Therefore, we assume that the actor type referred to in the challenge corresponds to either a position type or a role type. The class **Employee** is associated with the class **Person** via delegation, that is, an instance of **Employee** serves as a role of an instance of **Person**.

Further Issues: To understand the implications the model in Fig. 6 has on objects on lower levels it is pivotal to look at intrinsic properties in general and at intrinsic associations in particular. The impact of intrinsic attributes like the already mentioned ones of **CompOrgUnit** is obvious. The attribute **orgCriterion** determines whether a type of organizational unit is mainly devoted to a function, e. g. procurement, or to an object, such as "consumer electronics". Note that we do not account for other criteria to form organizational structures like matrix or tensor because that would considerably increase the complexity of the model without adding to the solution demanded for by the challenge.

With respect to model integrity and to the idea of multi-level language/model hierarchies it is important that dependencies between these associations are accounted for. The association **composedOf** can be defined for those classes only, the meta classes of which are linked through an instantiation of "subordinated". For example: a class like **Department** on level 2 may be linked to the **subsection** to define that subsections are subordinated to departments. As a consequence, concretizations of **Department**, like **MarketingDepartment** can be associated via "composedOf" only to classes that were defined as subordinated, e. g., **MarketResearch** as a concretization of **subsection**. These dependencies are expressed with constraints that are defined with the XOCL at the level of the generic model already, e. g. the constraint **properPart1** and the intrinsic constraint **properPart0** (see below). For a discussion of characteristic aspects of multi-level constraints see (Tony Clark and Ulrich Frank 2018).

```
context CompOrgUnit
```

```
  // Notice how these constraints use
  // 'of' and 'isDescendentOf' to
  // move down and up the multi-level
  // structure of the model.

  // properPart1 requires all CompOrgUnits
  // at level 1 to be composed of OrgUnits
  // at level 1 that are instances of the
  // corresponding superOrdinated type at
  // level 2.

@Constraint properPart1
   self.getOrgComposed1s()→forAll(i |
   self.of().getSubOrdinated()→exists(c |
     i.isDescendentOf(c)))
end

  // properPart0 requires all CompOrgUnits
  // at level 0 to be part of OrgUnits
  // at level 0 that are instances of the
  // corresponding composedOf type at
  // level 1.

@Constraint properPart0
   self.getIncludes0()→forAll(i |
   self.of().getOrgComposed1s()→exists(c
    | i.isDescendentOf(c)))
end
```

These constraints achieve two aims: (1) To require that the model on level 1 is consistent in terms of the type structure so that it is not possible to create a model with *missing classes*; (2) To require the structure of objects at level 0 to be type-wise consistent. Both of these aims are important when considering the semantics of the models and the ability of tools that support the use of the domain specific languages that are created.

Note that the associations between the classes **`OrgUnit`** and **`CompOrgUnit`** also define which position types may be assigned to which composed organizational unit. That is a consequence of applying the composite pattern. Usually there are no restrictions on L2, that is every position type on L2 might be subordinated to any composed organizational unit type on L2. On L1, however, it may be the case that certain position types must not be assigned to certain types of organizational units. Using multi-level constraints we achieve a model that constrains the assignment of position types at every level.

Further aspects could be accounted for that relate to specific requirements related, e. g., to human resources such as more elaborate conceptualizations of required skills or of performance indicators, or to accounting, like a more elaborate structure to represent wages. While these aspects are not subject of the challenge, a high level DSML for modeling organizational structures could include them to increase its potential utility. In addition, certain aspects of corporate governance, concerning, for example, decision making authority or responsibilities related to data and systems, could be represented.

As we mentioned already, all classes in the model inherit the attribute **`name`** from the top level meta-class **`Class`** in XCore. The attribute is not shown explicitly in the model to avoid complexity. Accordingly, the attribute **`lastModified`** was added to **`Class`**. The slot values are also not shown in the diagrams, but are accessible. In addition, the modification of the meta object protocol would allow for a more sophisticated solution which would make sure that every modification of an object's state results directly in updating the slot that corresponds to the attribute **`lastModified`**.

The analysis of position types on L2 in Fig. 7 may lead to the suspicion that design principle 4 was violated, because the distinction through the attributes **`focus`** and **`description`** may be seen as semantically poor. While we regard this as a valid objection, we believe that this additional level makes sense, because it allows for a more elaborate conceptualization of certain position types. For example technical positions (**`TechPosition`**) may be characterized by further, more specific properties that, within a certain concepts are common to all position types of this kind, e. g. "requiresEgineeringDegree" or "programmingSkills". Similarly, the class **`QM_Role`** could be associated with quality management certificates to support corresponding specifications with specific types of quality management roles. That would be in line with design principle 1.

4.2.2 Focus on Software Development Firm

According to typical textbook definitions, a business process type defines a schema that clearly

controls the execution of all business processes of the same kind. This is probably not the case for software development processes. They rather qualify as projects with respect to the fact that the concept of a project is often defined by the singularity of its nature. However, the design of a development process will usually not start anew for each project. Instead, it makes sense to define a process schema similar to that of a business process. However, different from a business process schema, it should allow for a higher degree of flexibility for its instances in order to account for the specific peculiarities projects have to deal with.

Core Concepts: The challenge explicitly mentions the following "actor" types in the software development firm: developer, analyst, senior analyst, senior manager, tester, and project leader. Except for tester (see below), they are all modeled as position types. Note that neither senior manager nor project leader are needed for modeling the software development process in accordance with the requirements. Instead of senior manager, we used the more context-specific type of senior project manager (see Fig. 8).

The attribute `creationDate` represents the date the corresponding position type was first introduced into a company. The class `SoftwareProject` is not mentioned in the challenge. We added it to the model, because it would be required to define the structural context a project is taking place in. Again, the classes used to define position types could be defined in a more differentiated way. The challenge also refers to particular actors Ann Smith and Bob Brown. Both are shown in Fig. 8.

Design Decisions: According to P15 an "*actor* may have more than one *actor type*." S11 illustrates this requirement with a specific instantiation: "*Bob Brown* is an *analyst* and *tester*." It is a common conceptualization suggested in most textbooks on organizational design that an employee fills one and only one position. This seems to be an iron rule in most organizations (even though, one may find exceptions). To not violate this rule and to satisfy the requirements at the same time we decided to model tester as a role. That would indirectly respond to requirement P15: an actor could fill one position but in addition multiple roles. S11 would be accounted for too, since L0 Bob Brown could be represented as holding the position of an analyst and the role of a tester at the same time.

It may seem more appropriate to model the attribute `salary` with `Employee` rather than with the class `Position`. Nevertheless, we decided for the second option because it allows for a higher level of abstraction and a more consistent handling of organizational governance. While it is common practice to define a salary range for a certain position type, and to assign a value out of this range to a particular position, it is not common to define a salary for an employee independent of the position he holds. The hierarchy of position types would allow the definition of a certain salary or a salary range for position types through attributes within their classes on L2. This could be done with employees only if they were represented through additional (meta) classes, too. However, we do not see how employees could be (intrinsicly) classified without referring to the positions they hold.

Further Issues: To further improve modeling productivity and to contribute to model integrity at the same time, one could integrate more specific position types that are known in the software industry. Also, the conceptualization of position types, role types and types of organizational units could be enriched with more specific properties, such as specific skills or required resources (hardware, software).

4.2.3 Focus on Insurance Company

Core Concepts: According to the challenge, the following position types in the insurance domain should be accounted for: claims handling manager, financial officer. These position types are created through concretizations of the classes `AdminManager` and `FinanceManager`. In addition, we introduce the position type `ClaimHandlingClerk` as a concretization of `ClericalPosition`, because we assume that process instances are not handled

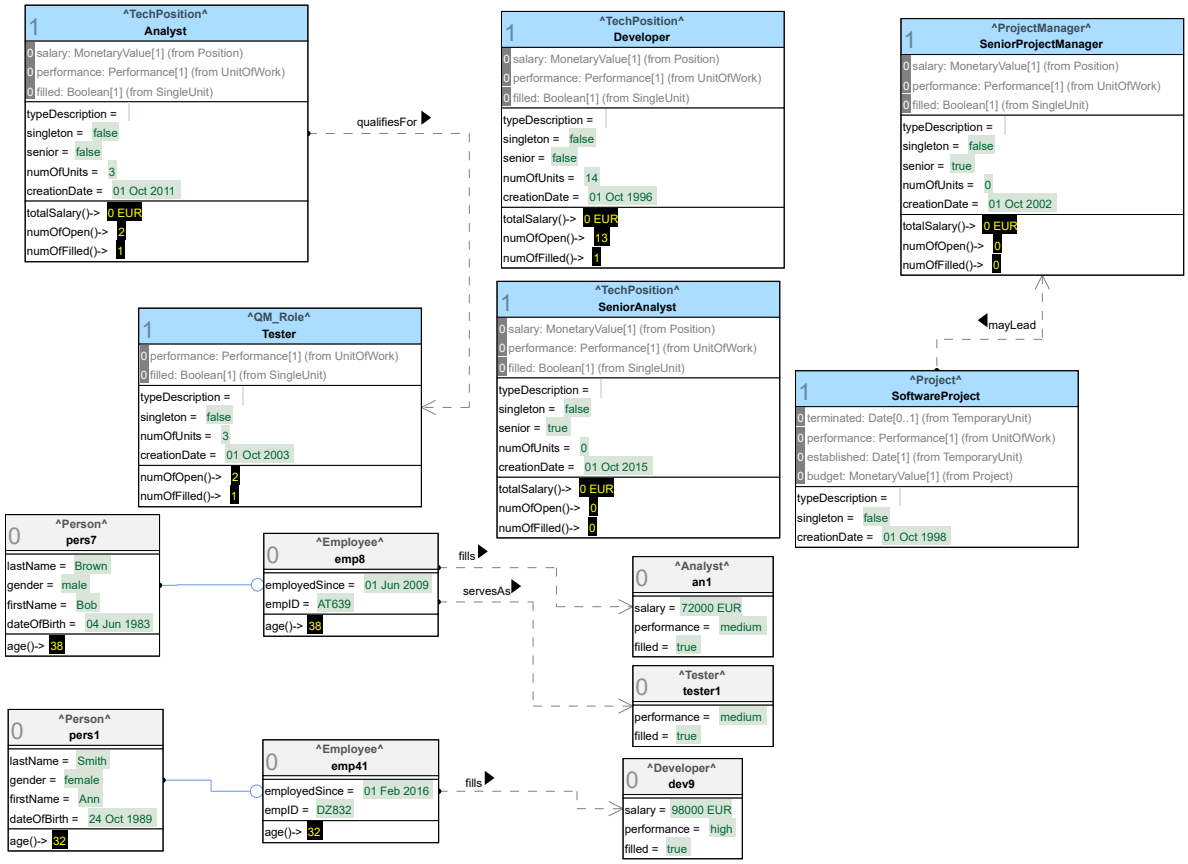

Figure 8: More specific concepts to model organizational units in sofware firms

by a manager. Fig. 9 shows the position types and corresponding instances.

Design Decisions: Since the basic structure of the model shown in Fig. 9 is widely in line with the corresponding model of the software development case, no further design decisions were required here. There was only need to decide for particular interpretations of the requirements. P6 refers to the actors *John Smith* and *Paul Alter*, which are allowed to assess claims. We assume that this allowance it based on the positions they hold. Therefore, we defined the position type **`Claim Assessor`**. According to P4) "*Ben Boss created* the task type *assess claim*". The challenge does not mention the position filled by Ben. We assume that somebody who fills a specific position is required for this task. Therefore, we introduced the position type **`ProcessDesigner`** and linked an instance of it to an instance of **`Employee`**, which is in turn linked as a delegator to the object that represents Bob Brown.

Note that the values returned by operations like **`totalSalary()`** or **`numOfOpen()`** in position types on L1 or **`age()`** in objects on L0 serve to demonstrate that models created with the FMML^{x} in the XModeler^{ML} are executable. The code below shows the implementation of the intrinsic operation **`totalSalary()`** within the class **`Position`** on L3.

```
currency := Currency("EUR","EUR",1);

context Position
```

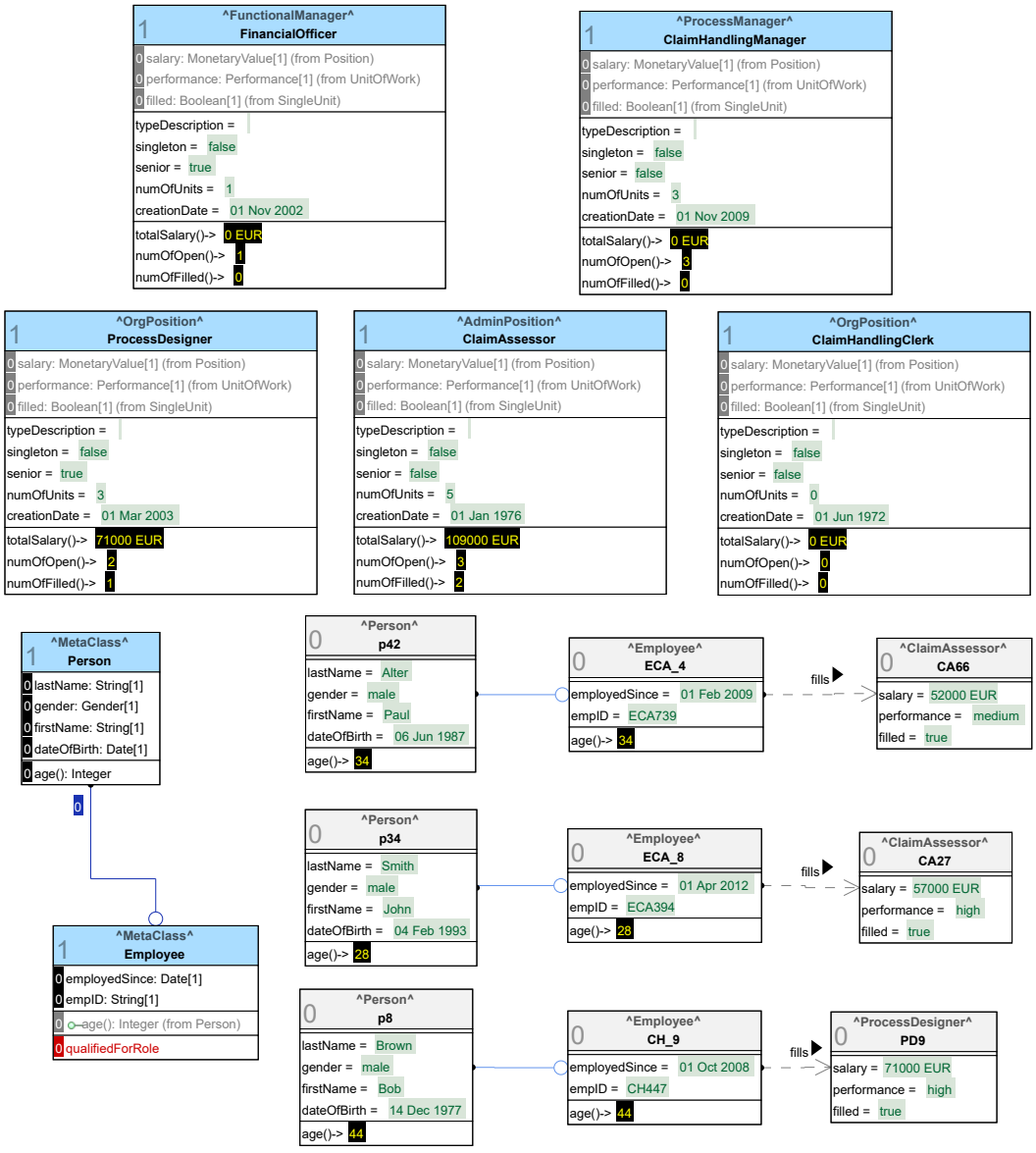

Figure 9: Positions required for the insurance case

```
@Operation totalSalary [monitor=true] ():
    MonetaryValue
    let sum = 0
    in @For i in allInstances do
         if i.filled
         then sum := sum + i.salary.amount
         else false
         end
       end;
       // return an instance of the class
       // MonetaryValue value using the
       // currency Euro.
       MonetaryValue(sum,currency);
    end
  end
```

Like the diagram excerpt in Figs. 8 and 9 demonstrates the use of delegation in the XModeler^{ML} . An object that is concretized or instantiated from a class that is defined as a delegator class by a delegation association, like **Employee**, delegates those methods that are not included in the protocol of its class transparently to the object of the corresponding delegatee class it is linked to. Note that delegation is different from inheritance, because it allows "inheriting" the state of an object, too. In that sense, it is suited to mitigate problems that arise from the fact that object-oriented languages allow an object to be of one class only (for a detailed discussion see (Frank 2000)).

4.3 Artifacts

The artifacts used and produced in the processes that are subject of the challenge are objects and, more specifically, documents. Different from organizational design, both case descriptions do not share noteworthy commonalities apart from a rather abstract general notion of resource. However, we assume that there are indeed more commonalities that could be accounted for in a generic model of resources used and created within business processes. First, both cases are likely to use IT resources such as application systems, tools, platforms etc. Second, software development processes may refer to documents other than software artifacts, including contracts. However, we decided to focus on the requirements specified in the challenge and not to extend the scope too much. As a consequence, the "Generic Artifacts" model in Fig. 5 is restricted to the class **Resource** that is referred to by the generic process model.

4.3.1 Focus on Software Development

Core Concepts: The classes that represent artifacts in the software domain are shown in Fig. 10.The model comprises three kinds of artifacts, represented by the class **SoftwareArtifact**: code, models and reports. We assume that all artifacts are created with a language. This is expressed through associations between artifact types and language types.

Design Decisions: It is not trivial to determine the nature of software development artifacts. On the one hand, they can be regarded as linguistic artifacts. A model, for example, can be conceptualized as being created with or instantiated from a modeling language. However, from a project management perspective, it can be useful to regard a model as a specific kind of document. We decided to account for both aspects.

We regard all documents that are created to analyze, design, implement or test a software system as *software artifacts*. The general concept of a software artifact is represented by the class **SoftwareArtifact** on L3 (see Fig. 10). At this level, we know already that software artifacts on L0 have a creation date, a state, e. g., "early", "in progress", "complete", that they may be write protected, etc. The concretizations of **SoftwareArtifact** on L2 represent different kinds of artifacts, such as code, models, reports, etc. and associations between these. For example, we know that every artifact may depend on other artifacts. In addition, every artifact may require a language for its construction. This knowledge is represented by the association **requiresLang** between the class **Resource** and the class **Language** and the circular association **requires** of the class **SoftwareArtifact**, both on L3. In addition, the intrinsic associations **maybeWrittenIn** and **writtenIn** between **Resource** and **Language**, to be instantiated on L1 and L0 respectively, allow assigning a particular artifact, like, e. g., a COBOL program to the language it was written in. The association **requiresLang** is to constrain the range of languages

that can be used to specify **maybeWrittenIn** links which in turn constrain **writtenIn** links. Both constraints are expressed in **Resource**. The constraint **appropriateLangKind**, which is to be checked on L0 is shown below

```
context Resource
@Constraint appropriateLang
   self.of().getLanguages()→exists(c |
   self.getWrittenInLang().isDescendentOf(
    c))
end
```

The association **evaluatedBy** between the classes **Code** and **TestReport** addresses requirement S9: "Each *tested artifact* must be associated to its *test report*." Fig. 10 also illustrates how the XModeler^{ML} supports the construction of models on lower levels. In the shown example, the cardinality constraint implicitly defined with the intrinsic association **writtenIn** is violated. As a consequence, the user is offered a set of languages, the model could be linked to.

Software artifacts are created with a language. While natural language plays an important role in that respect, e. g. for documents produced during requirements engineering, we do not explicitly account for natural language. First, it is likely to be part of many software development documents anyway, e. g. in comments. Second, we assume that every document used professionally in a software development process reflects some kind of structure. A manual, e. g., should comply with a default structure for representing manuals. We regard this structure, which could be defined, e. g., as an XML DTD, as the language, a corresponding document is created with.

Common properties of all languages used to create software development artifacts are represented by the class **Language** on L3. It includes various attributes that enable the concretization of a range of more specific kinds of languages on L2. These include, e. g., **executable** to indicate whether a class of executable languages is represented, or **dsml** that allows to specify whether a language is conceptualized as a DSML. Classes on L2 serve the representation of programming languages (**ProgLang**), modeling languages (**ModLang**), and further classes which can be added through a characteristic instantiation of the attributes provided for this purpose in **Language**. Accordingly, classes on L2 can be concretized into language classes on L1 through the instantiation of attributes like, e. g., **graphicalNotation** or **isDSL**. In addition modeling languages can be classified as static, functional, or dynamic. Finally, objects on L0 represent particular languages such as COBOL. Languages can be associated with software artifacts as long as the corresponding constraints are not violated.

Further Aspects: Like all multi-level models that we present, the model of software artifacts is executable and, thus, may serve as an information system that can answer questions related to software artifacts and their use, e. g. with respect to planning and staffing projects. To provide more support, further relevant aspects could be included, such as the tools that are required and/or are available to create and manage software artifacts, and the platforms they run on, or dependencies between the artifacts.

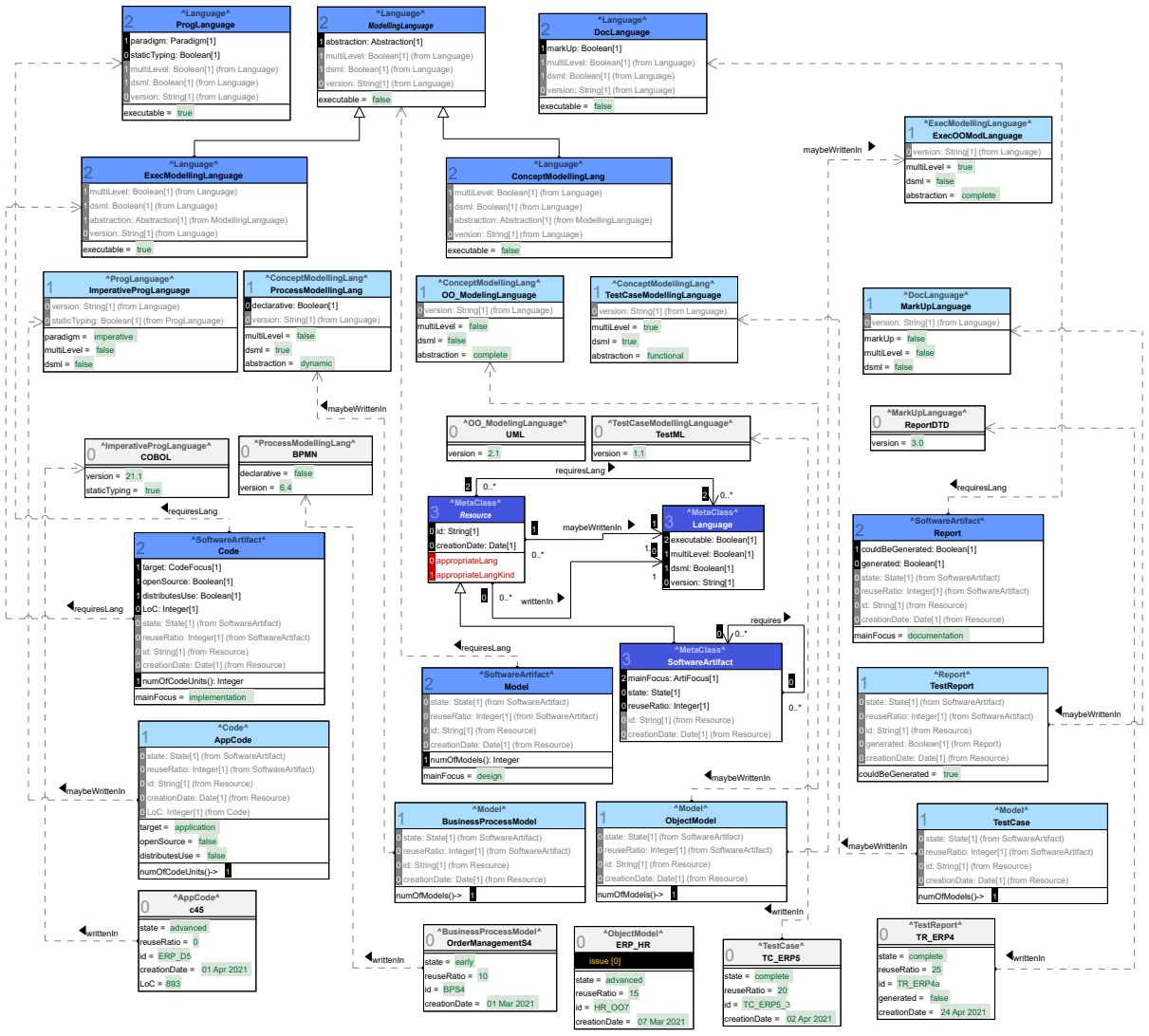

Figure 10: Artifacts and languages required for the software development process

4.3.2 Focus on Claims Handling

Core Concepts: Different from the software development process, the challenge lacks a detailed description of the claims handling process. Therefore, we had to make a few assumptions concerning the resources required to run the process. First, the process requires some kind of a claim made by a policy holder. We assume that a claim is filed electronically, e. g. through a web page and is represented by an object. Second, the process requires access to the corresponding contract, which is a specific kind of document. Finally, some kind of compensation statement or the denial of compensation are created. They are official messages that need to be documented accordingly. The model shown in Fig. 11 comprises the classes needed to represent these concepts.

The class **`Document`** on L3 serves the representation of knowledge about any kind of documents in the insurance domain (and maybe beyond that). It can be concretized into the class **`Contract`** on L2 that defines additional properties relevant for contracts. A further concretization that is relevant for the challenge is the class **`CustomerNotifcation`**

that defines the properties required for official customer notifications. It can be concretized into the class `ClaimsHandlingNotificiation` on L1 which itself can be concretized into objects that represent particular notifcations such as granting or denial of a claim for damages. The class `Contract` can be concretized into classes that represent specific types of contracts, such as some kind of damage insurance, which itself can be instantiated into a particular damage insurance contract.

Design Decisions: The challenge mentions policy holder as an actor. We represent a policy holder through the class `PolicyHolder`. It could be concretized from a higher level class such as `PrivateCustomer`. However, we decided against this option, because there would be no use of this additional abstraction within the scope of the challenge. Hence, a policy holder is represented as a role, therefore it needs to be linked to an object that represents the corresponding person. The intrinsic association `holds` between `PolicyHolder` on L1 and `Contract` on L2 represents the knowledge that every contract in the domain requires the participation of a policy holder. This conceptualization also allows representing employees who hold policies without redundancy.

4.4 Processes

The design of the two process types that are subject of the challenge require the use of a process modeling language. Therefore, we will first outline a meta model that defines basic concepts of such a language. A comprehensive specification of a process modeling language would clearly exceed the scope of a journal paper. The BPMN specification (OMG 2013), for example, comprises about 500 pages. The specification of the process modeling language, which is part of the MEMO-OrgML (Frank 2011b) that serves us as a reference still comprises about 120 pages. Therefore, the (meta) model presented in Fig. 12 is restricted to those aspects that need to be covered in order to satisfy the requirements described in the challenge. While this model qualifies as a multi-level model, because it includes classes on more than one level, the additional abstraction enabled by the FMML^x is restricted to static and functional aspects and does not account for multiple levels of dynamic abstractions. Subsequently we present thoughts on the design of multi-level process models which go beyond the challenge.

4.4.1 Generic Process Model

Core Concepts: The objects that constitute the process itself are specified in the multi-level process model in Fig. 12. It is based on a reconstruction of parts of the metamodel of the MEMO-OrgML (Frank 2011b), which was defined using a traditional, two level language hierarchy. The diagram shown in Fig. 12 represents an excerpt of the upper level classes of the entire model only. It should be sufficient to illustrate that it is suited to satisfy the requirements described in the challenge. The class `Event` is on L3. This allows for the definition of specific language concepts to represent start and stop events on level 2. The model also indicates how activity types are related to organizational role or position types (in the challenge accounted for on a different level of abstraction as "actors") and to artifacts types. The intrinsic association `mayPerform` between `SingleUnit` and `ProcessActivity` allows for the definition of position and role types that are qualified to perform activities of a certain type. The additional intrinsic association `performs` allows to control the assignment of particular positions or roles to an activity (see below).

Control flow concepts comprise *sequence, branching, fork and synchronizer*. Branchings indicate that a decision has to be made on how to continue a process, which requires the specification of at least two alternative branches. For each decision type, it can be specified to what degree it is automated. Forks serve splitting the control flow into a number of concurrent threads. Synchronizers are used to represent the specific kind of synchronizing concurrent threads. An OR synchronizer defines that the concurrent threads are synchronized as soon as the first one of the respective threads terminates. In contrast, the use of an AND synchronizer indicates that all

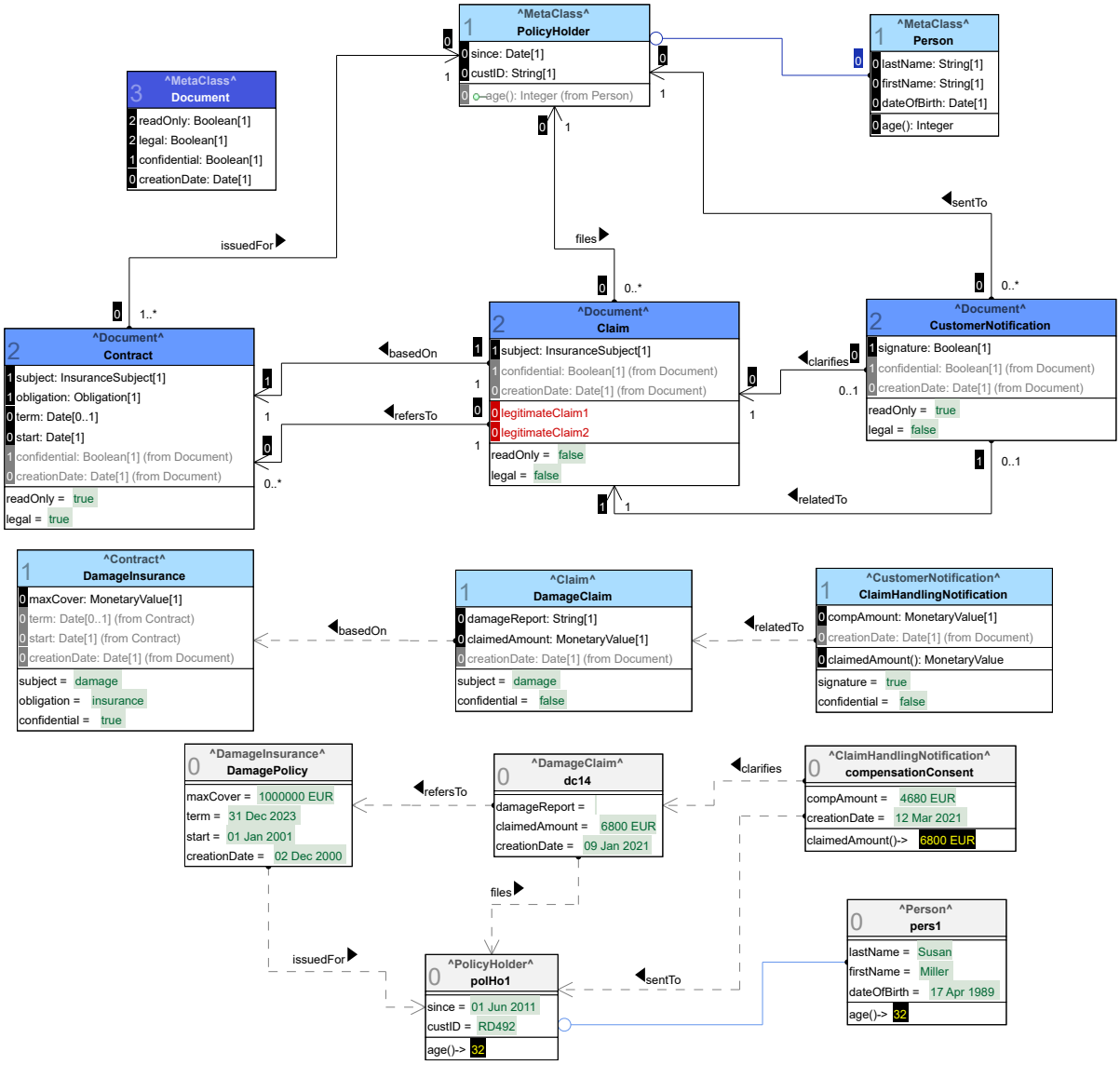

Figure 11: Artifacts required for the claim handling process

related threads have to terminate before the process may continue. Note that we do not use the term "gateway" in our model. We regard it as too is ambiguous and, hence, potentially misleading, because it is used for essentially different control flow concepts. This ambiguity is clearly shown in the "gateway class diagram" in the BPMN specification (OMG 2013, p. 88).

Design Decisions: Events between processes are not mandatory in some business process modelling languages such as the BPMN. Enforcing the use of events may sometimes be perceived as inappropriate and result in seemingly redundant events, e. g. "coding complete" produced by the activity "coding". However, events are of pivotal relevance for managing the control flow. Without some kind of event it would, e. g., not be possible to detect the end of an activity. Therefore, we decided that the proposed language requires

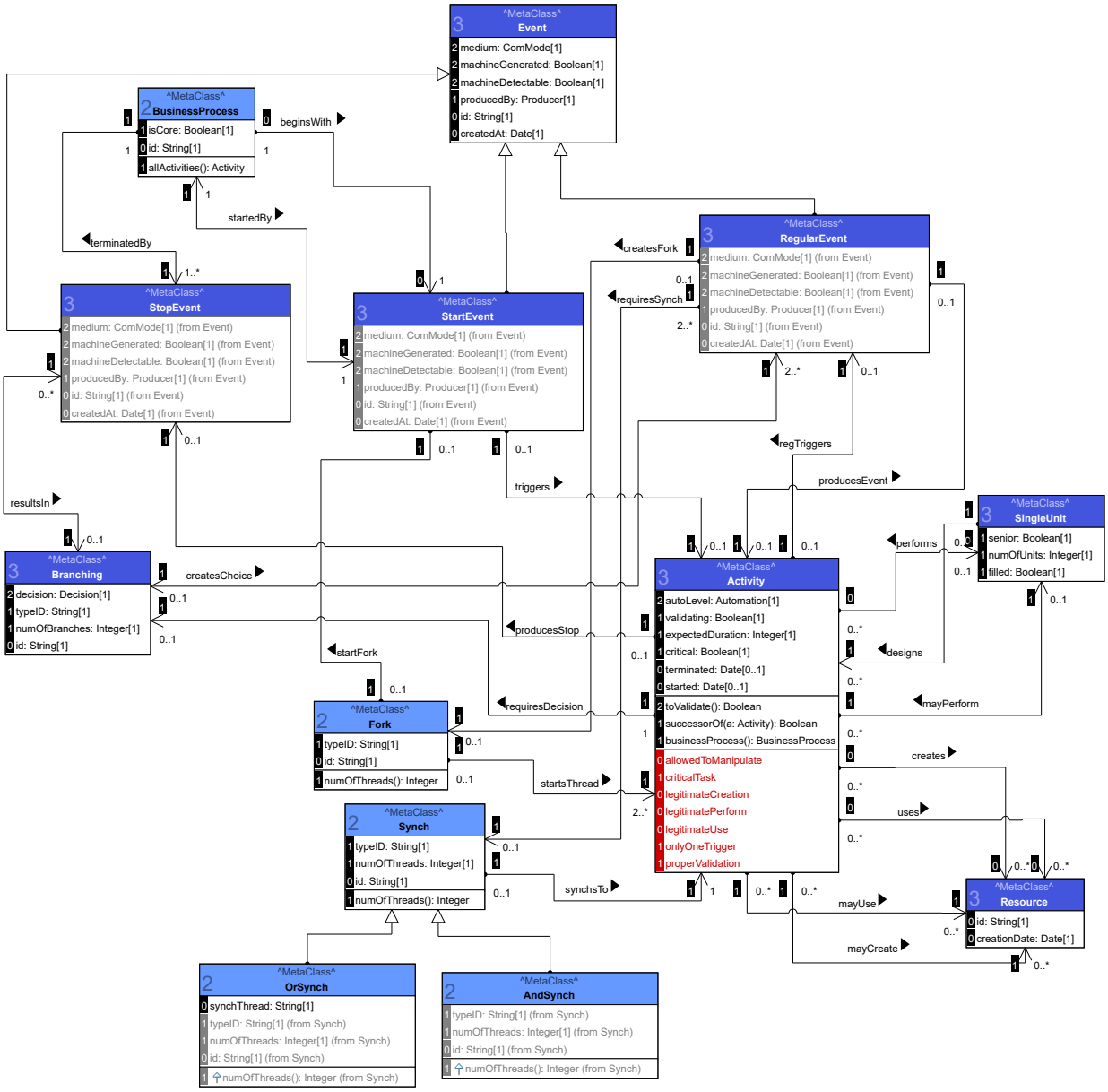

Figure 12: Generic process meta model

the specification of events before and after every activity.

In general the design of a multi-level model presents design decisions regarding the choice of level to place a certain concept. In the case of a generic process model that relates especially to the question whether core concepts such as event or activity should be placed at L2 or at L3. Placing them on L3 allows concretizations to classes on L2 that represent specific kinds of activities, such as fully automated, partially automated or manual. We assume that it makes sense to distinguish these types, because that allows for the definition of specific constraints, e. g. that fully automated activities must not be performed by a human. This decision corresponds to design principle 1, which suggests to model that knowledge on the highest level and not only on the level of specific process

types. For the same reason, we decided to define start and stop events already on L3, too. As a consequence, L2 may comprise an extensive range of activity (meta) types comprising, e. g., machine generated start events or machine generated start events that are not machine detectable, etc. It is important to note that this extensive use of different classes creates a challenge for the implementation of modeling tools. During the course of creating a model, it may, for example, happen that an activity that was at first modeled as fully automated needs to be changed to partially automated, which would imply some kind of class migration. Therefore, it also depends on the change operations offered by the corresponding modeling tool, whether or not to decide for additional (meta) classes on L2.

In addition to activities and events there is need for control flow concepts like sequence, branchings and concurrent threads. Concurrent threads are enabled ty the classes **Fork** and the subclasses of the abstract class **Synchronizer**. The class **Branching** serves the representation of branchings. It may be regarded as inappropriate to model **Fork** and synchronizers on L2 and **Branching** even on L3, because it seems they are instantiated only once. Nevertheless, we decided otherwise. The class **Branching** on L3 allows for concretizations on L2 that represent certain kinds of decisions such as "automated", "human based on clear rules", etc. Hence, related constraints, for example, that a fully automated activity must not result in a branching that requires a human decision, can be defined on that level already. On L1 it is possible to define branching types that are characterized by a unique id, because branching types do not have a name. The number of branches, which is represented by an attribute could as well be computed by an operation. At level 0, that is at the level where a process is executed, the branching is collapsed into a sequence, but nevertheless, a descendant of **Branching** on L0 may be useful to store the corresponding decision. Similarly **Fork** and the subclasses of **Synchronizer** are concretized into classes on L1 that are characterized by a unique id. On L0 they serve the representation of particular process executions.

There is no explicit concept for modeling sequences. Instead, a sequence is implicitly modeled by various associations between event types, activity types and other control flow concepts. At the level of a particular process type, a sequence if represented as a link between instances of these elements (cf. the process diagram in Fig. 15). At the level of a particular process instance, there are no links anymore. However, that does not cause a loss of information, because the control flow of a process instances can be created from the one specified for its type and additional information provided by descendants of the classes **Branching**, **Fork**, and of subclasses of the **Synchronizer**.

Requirements P4 and S11 are related to the design of task types. They do not specify the preconditions an actor needs to satisfy to be allowed to design a task type. We decided for two supplementary approaches to account for this issue. First, the association **defines** between certain position types on L1 and the class **BusinessProcess** on L2 allows representing that holders of positions of a certain type are entitled to design an entire business process model. Fig. 13 shows a corresponding use of this association together with an example instantiation. Second, it is possible to specify that certain position types or roles types are eligible to design particular task types, which is enabled by the association **designs** between the classes **SingleUnit** and **Activity** (cf. Fig. 12).

In addition to the specification of the control flow, the proper execution of business processes calls for some kind of process governance. At the level of a generic process model, governance comprises rules that relate to the rights of those who participate in a process. Corresponding regulations are specified through various intrinsic associations between the classes **Activity** and SingleUnit. The association **mayPerform** defines the position types that are entitled to perform tasks of a certain type. The association **performs** is restricted to those roles and positions the types of which are linked to the type of the activity they are linked to. In addition, the right to perform activities of a certain type is also related to requirement P18. A task that may be performed by

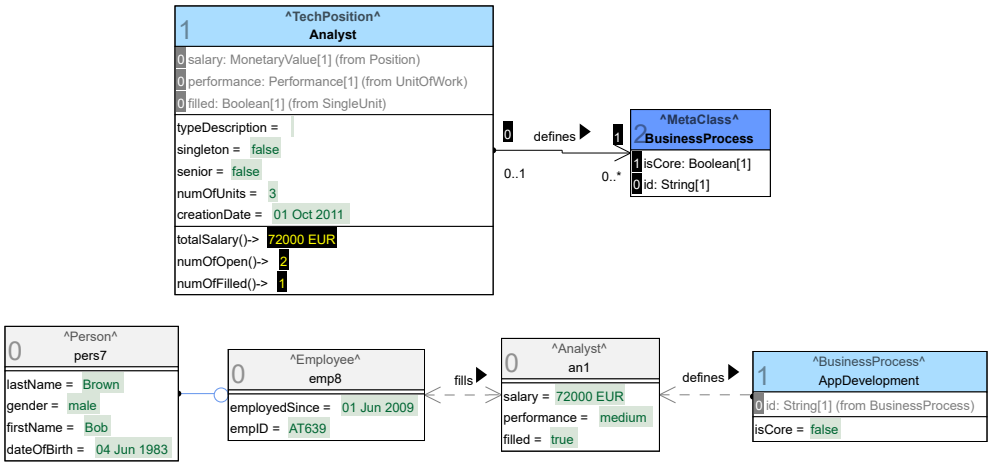

Figure 13: Representation of task type designers

the holder of a certain position can be performed by his superior as well. As we argued in Sect. 3 already, we regard it as problematic expressing that by specializing a management position type from a position type. Nevertheless, we wanted to account for the basic idea, namely, that a manager, as a default, has the right to perform all tasks, his subordinates are entitled to perform.

Both restrictions are represented by the constraint **legitimatePerform**. It checks whether the class of a performing unit at L0 is linked to the corresponding activity type through an instance of the association **mayPerform**.

```
context Activity
@Constraint legitimatePerform
   self.of().getQualiSingleUnit()→exists(
    c |
   self.getActUnit().isDescendentOf(c))
end
```

Note, however, that satisfying this constraint alone may not be sufficient to qualify a manager for performing a task. As we outline in Sect. 3 already, it would be problematic to grant a manager automatically the right to manipulate every resource her superiors may manipulate. Therefore, we specified a further constraint that accounts for specific skills which may be required for performing an activity (see constraint **allowedToManipulate** below).

Other aspects of process governance relate to the resources that are required for performing a process. The constraint **legitimateCreation** for instance, serves to check whether resources may be created within an activity. To this end, it checks whether the type of the activity and the types of the corresponding resources are linked by an instance of the intrinsic association **mayCreate**:

```
context Activity
@Constraint legitimateCreation
   self.getCreatedResources()→forAll(i |
   self.of().getPossCreatedRess()→exists(
    c | i.isDescendentOf(c)))
end
```

According to requirement P9, tasks the type of which is characterized as critical may be performed by “senior actors” only. That leads to the question how a “senior actor” should be modelled. We decided to use position types for that purpose. Each position type may be specified as senior. We believe that this approach is more appropriate than representing seniority with the class **Employee** or even with the class **Person**, because it reflects some kind of authority that is formally linked to the position within the organizational hierarchy. The constraint **criticalTask** is to check whether the type of a position or role assigned to a critical task type is qualified as senior. Note that the constraint is simplified in the sense that it does not check whether a unit had been assigned yet.

Similar simplifications apply to other constraints as well.

```
context Activity
@Constraint criticalTask
   self.getCritical() ⇒ self.
    getQualiSingleUnit().getSenior()
end
```

In addition, requirement P9 demands that a critical task type which creates a certain artifact type needs to be followed by a corresponding validation activity. To account for this requirement, we decided for various extensions. We added the operation **allActivities()** to the class **BusinessProcess**. It returns the set of all task types included in a process on L1. Then we added the intrinsic attribute **validating** to **Activity**, which allows specifying that an activity type is suited to validate the outcome of a previous activity type. The operation **toValidate** of **Activity** computes whether the outcome of an activity requires validation — by checking whether the activity is critical and a resource was created. Finally, the operation **successorOf(a: Activity)** computes whether an activity type is succeeding another activity type, by accessing the set of all activity types through the methods **businessProcess()** and **allActivities**. These extensions allow the definition of the constraint **properValidation** to **Activity**:

```
context Activity
@Constraint properValidation
   self.toValidate() ⇒ self.
    businessProcess().allActivities()→
    exists(i | i.successorOf(self) and i.
    validating)
end
```

Note that this constraint is not entirely satisfactory. First, there may be more than one task type that need validation. That would require additional information on how to identify the corresponding validating task type. The challenge does not provide such information. Second, it can only be determined on completion of a process instance at L0 whether an actual validation took place. A process type may include a branching with only one alternative that includes a validation task. At the level of a process type it cannot be decided whether this path will be taken during the execution of a process.

The intrinsic associations **mayReportTo** between **Position** and **ManagementPosition**, to be instantiated on L1, and **reportsTo** (or "supervisedBy" in the opposite reading direction), which is instantiated on L0, serve accounting for P18. This conceptualization allows expressing the basic idea of P18 and avoids at the same time problematic consequences that may occur with the suggested specialization relationship. If, e. g., a developer is allowed to modify the code she works on, it does not necessarily mean that her superior is allowed to do so as well. However, the superior is likely to have the authority to demand a modification.

Further issues: It may appear at first sight that the association **performs** (and, accordingly, the associations **mayUse** and **mayCreate**) is redundant. If the association **mayPerform** was instantiated on L0 only, then positions could be assigned to activities. However, in that case, one could not express that only certain types of positions or role qualify for that particular activity type, which would be a serious threat to process integrity.

The consistent use of concurrent threads requires rather complex constraints that ensure proper synchronization. For a corresponding constraint see (Frank 2011b). We did not add synchronization constraints to the model presented here, because they are not required by the challenge and would substantially increase the complexity of the model.

As we shall see below, the multi-level process model allows for instantiations on level 0. However, the proper creation of process instances on L0 requires dynamic instantiation (see illustration of specific problem in Fig. 18) as it could, e. g., be provided by some form of process execution engine .

While defining a task type within a business process as critical is definitely better than leaving that characterization to particular tasks at L0 only (design principle 1), the requirements related to projects may be different. A certain reference process model for software development projects

may, e. g., include the activity coding, but it may not be regarded as critical in all projects. In those cases, it would be more appropriate to allow for defining a task as critical at L0 only. Note that this remark relates to the general concern that processes used for managing projects are different from business processes and may, therefore, recommend a different kind of conceptualization.

Process modeling includes further aspects that go beyond the requirements described in the challenge. However, with respect to preparing for process execution they need to be accounted for. First, process governance will often include rules that concern the identity of actors who manage tasks and of corresponding resources as well. For example, the position type "sales assistant" may be assigned to various activity types of an order management process. Also, every activity type may create or use certain resource types. A more specific process governance may define that for every instance of the order management process all corresponding tasks have to be assigned the same instance of the position "sales assistant". This could be achieved by adding a constraint to more specific process models. Since such a constraint is not too unusual, it would be beneficial representing the corresponding knowledge at the level of the generic process model already. That would, however, require to use some kind of optional constraint template that could be further refined at the level where the missing knowledge is available.

Similarly, it may be necessary to express that the instance of a certain resource type on L0 has to be unique during the entire process, for example, an order or a contract. Again, this could be expressed through constraints at the level where this restriction applies or at a higher level with an optional constraint template. With respect to information resources such as documents or objects in general, information flow has to be accounted for, too. It makes a difference, whether an object's state is passed through a process, or only a reference to the object. Therefore, it should be possible to express that together with the used communication channels. With respect to process analysis further aspects may have to be accounted for, such as costs, failures, bottlenecks, etc.

4.4.2 The Software Development Process

Before we present the multi-level process model, the software development process as it is defined in the challenge is represented in the notation of the MEMO-OrgML (Organisation Modeling Language) (Frank 2011a), (Frank 2011b), because a specific notation allows for a more comprehensible representation (Fig. 14). The model is based on the UML activity diagram in Fig. 1 of the challenge description. In addition, it also represents the relevant context with respect to organizational units and artifacts. This way, it served us as a blueprint for developing the corresponding FMML^x model.

4.4.3 Software Development Process

Core Concepts: The objects the process tasks refer to, that is, the relevant context, are defined in the models depicted in figures 6 and 10. The designators used for naming the artifacts used or produced by a process task correspond to the names of the classes in the multi-level models of artifacts for the software development domain (Fig. 10). Accordingly, the designators that indicate the role or position types, the instances of which are responsible for a process task, correspond to the classes with the same name in the multi-level model of the organizational structure in the same domain.

Fig. 15 and 16 depict the model of the software development process created with the FMML^x .

Design decisions: According to S7, *Ann Smith* is "the only one allowed to perform coding in COBOL." As we argued already in Sect. 3, it would violate principles of organizational design to create a "lex Ann Smith". Therefore we represent the underlying idea, i. e., that only a person who is qualified to modify code is allowed to do so, on a more abstract level. The association **`masters`** between **`Employee`** and **`Language`** serves expressing that the corresponding employee is qualified to manipulate artifacts written in one of the languages linked to her. Since we assume that software artifacts are manipulated only within processes, the rule that only a qualified employee,

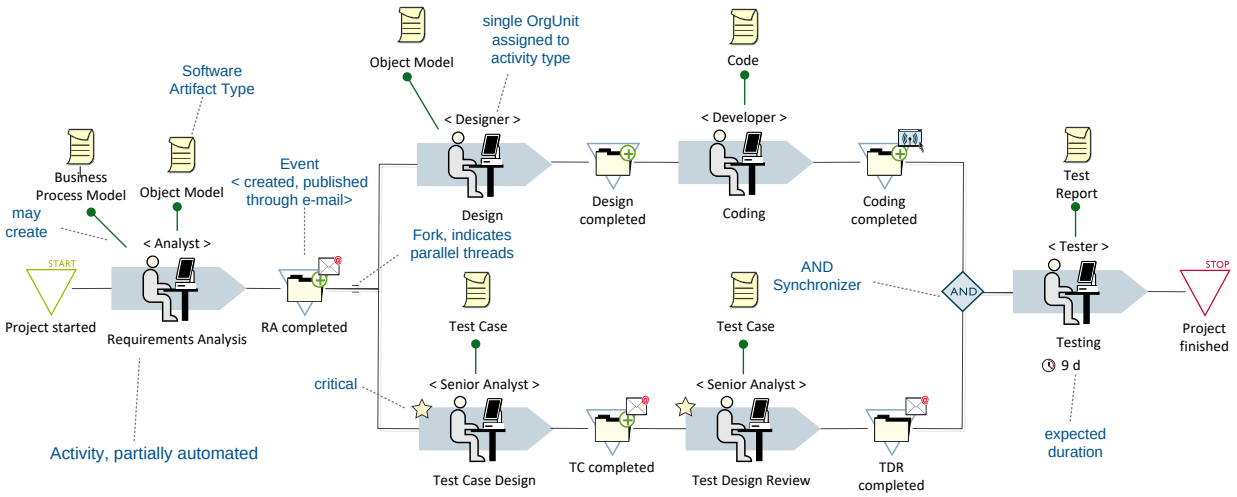

Figure 14: Diagram of the software engineering process model in the MEMO OrgML notation

i. e. one who masters the required language, may manipulate an artifact is implemented as a constraint within the software development process, namely the constraint **`allowedToManipulate`** with the class **`Activity`** (see Fig. 12).

Note that we decided to specify this constraint with the generic process model for simplification purposes only. It does not apply to the insurance case. Therefore, the activities specified for the software development case should have been concretized from a more specific class that could be specialized from the class **`Activity`** on L3. This class could then be used to specify the constraint.

```
context Activity
@Constraint allowedToManipulate
  self.getCreatedResources()→forAll(r |
      r.getRequiredLang()→forAll(l |
        getActUnit().getEmployee().
    getLangMastered()→includes(l)))
end
```

Fig. 17 illustrates the application of the constraint, which fails in the case of an employee who is in charge of an activity that requires a language he does not master. It would, of course, be possible to assign a responsible employee or position respectively to each artifact. However, that seems to be too restrictive, because it would require that particular employee to participate in all projects where this artifact is manipulated. That would create serious problems in case the corresponding employee is temporarily not available.

Further issues: From a static perspective, the concepts used to specify generic process models allow for representing particular process instances on L0. However, this kind of instantiation may produce results that are disturbing. The example of a partial instantiation of the software development process shown in Fig. 18 illustrates the problem. The instance of the class **`ObjectModel`** that is used during design should be the same that is created during requirements analysis. While it is characterized by the state "early" in the requirements analysis task, its state turns to "advanced" in the design task. However, with a static view on instantiation, it is not possible to represent two different states of an object, because an object is represented only once. If its state is changed later in a process that would cause a problem, because then the previous activity would be linked to a wrong state of the object.

To cope with this problem, one could use the workaround shown in Fig. 18: two different objects are assigned to the tasks in order to represent two different states. The fact that both objects represent the same object is expressed through a common identifier. There may be certain analysis scenarios where the use of such a workaround is helpful. However, in general, the instantiation of

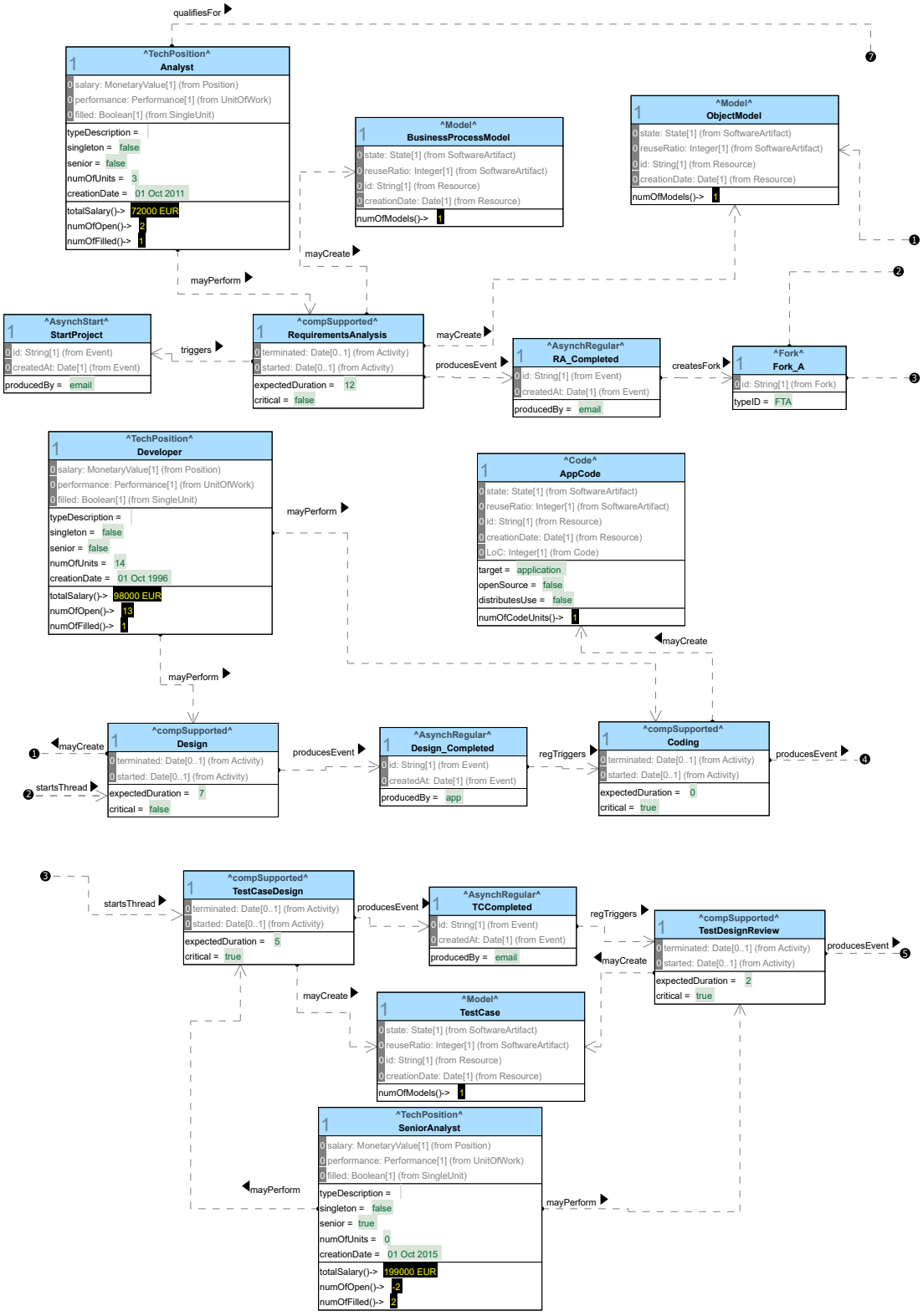

Figure 15: FMML^x model of the software development process and its relevant context (part 1)

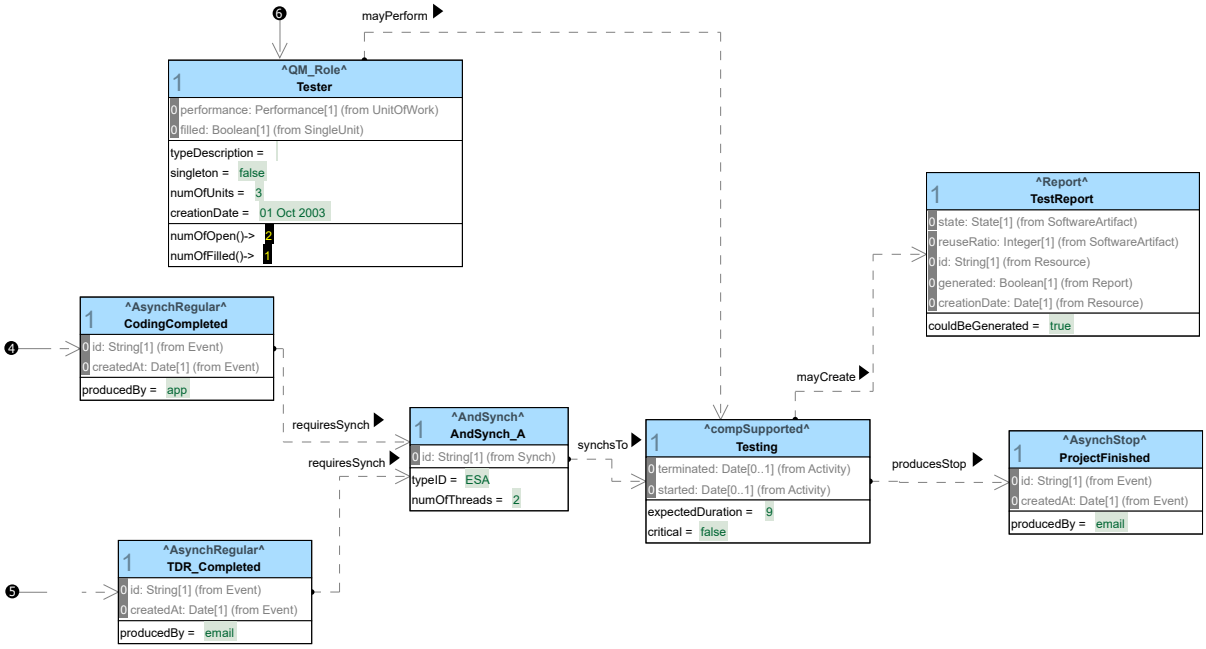

Figure 16: FMML^x model of the software development process and its relevant context (part 2)

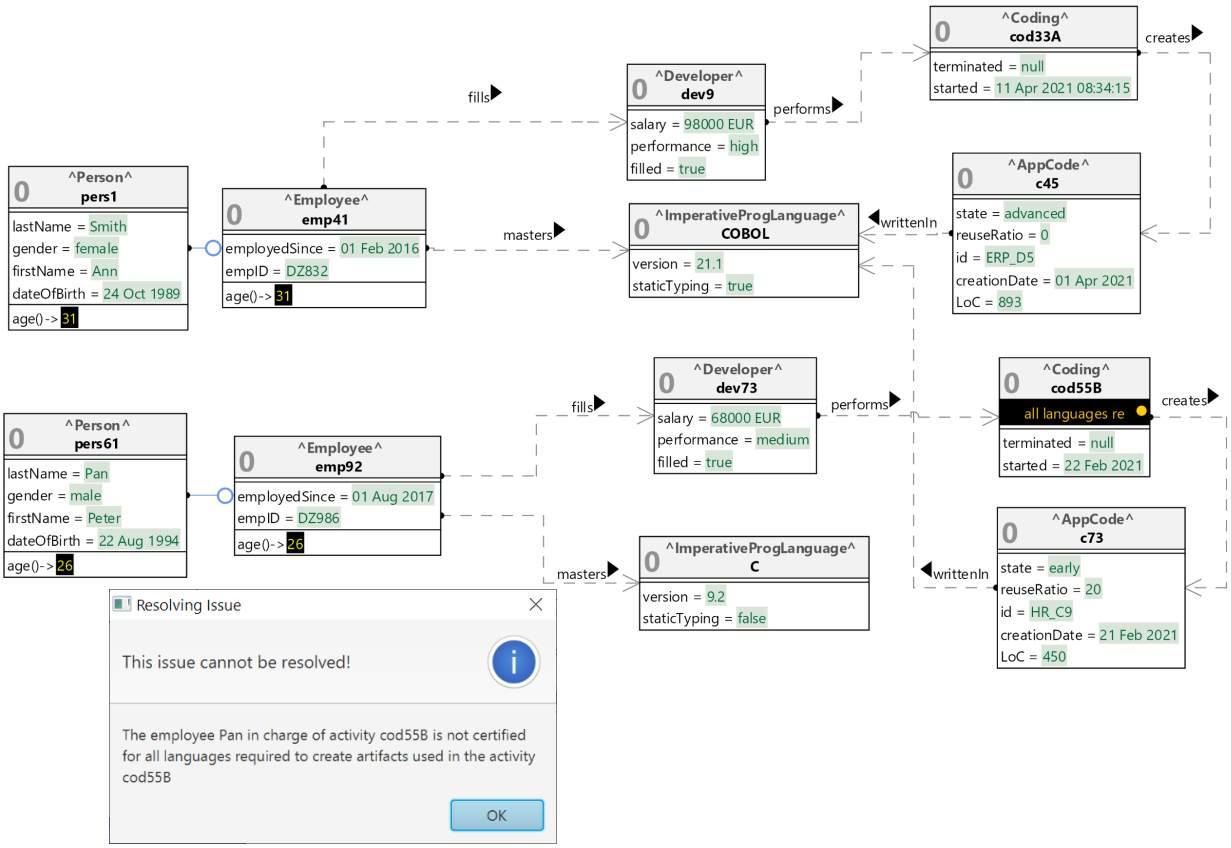

Figure 17: Abstraction of the "lex Smith" – for the classes **`Person`** *and* **`Employee`** *see Fig. 6*

a business process needs to be done dynamically. This allows for changing the state of objects during the course of instantiating activities step by step. The realization of the dynamic instantiation of a process type requires a process execution method in the end, implemented, e. g. in process management class, that is able to instantiate activities and to listen to the events that are created upon the termination of activities. Our model does not implement this kind of dynamic instantiation, but it could be extended with a process execution facility, e. g. to enable simulations.

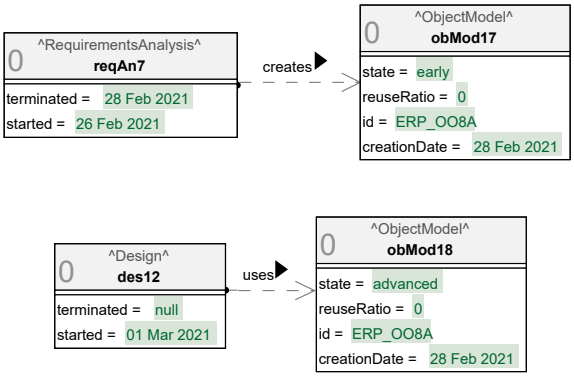

Figure 18: Illustration of problem with static instantiation of process types

4.4.4 The Claims Handling Process

The diagram in Fig. 19 illustrates our interpretation of the claims handling process that builds the foundation for the model that we will subsequently present. After a claim was received, the first activity, "Check Claim" is to check, whether the claim is formally covered by a valid insurance policy. We assume that a position of the type "Claims Handling Clerk" is in charge of this validation, the possible outcomes of which are reflected by the following branches represents this validation. If a valid insurance policy exists, the claim assessment activity is triggered. It is handled by an employee who holds the position of a claims handling manager and uses the insurance policy and the damage claim already used in the previous activity. The assessment of a claim includes determining the amount to be reimbursed. The final activity in the process, "Authorize Payment" is performed by either a financial officer or a claims handling manager.

Core Concepts: The construction of the claims handling process with the FMML^x takes the diagram in Fig. 19 as a blueprint. The document types used in the process correspond to the classes specified in the model of artifacts in the insurance domain (cf. Fig. 11). Apart from documents of the class **`ClaimHandlingNotification`**, they are only used, not created. The capability of an activity type to create or use a certain type of document is specified in the generic process model by the associations **`mayCreate`** and **`mayUse`**, which are instantiated as links in the claims handling process, which is shown in Fig. 20.

Design Decisions: The design of the process model did not include any specific design decision different from those required for the design of the software development process.

Further Issues: It seems obvious that a claims handling process may not include more than one instance of an insurance policy. However, the generic process model does not define a limit for the maximum multiplicity. Hence, more than one instances of insurance policy could be assigned to a task on L0. This issue could be addressed by a constraint either added to the corresponding instance of the class **`BusinessProcess`** on L1. However, that would create a challenge to model integrity, because it would partly contradict the corresponding assertion in the generic process model. While this was already known with the design of the generic model, the FMML^x does not offer a language concept to express that the maximum multiplicity may change within the concretization subtree.

5 Evaluation

The evaluation of the solution consists of two parts. At first, we will check the presented models against the requirements published with the challenge. The second part is focused on a more general comparison against traditional modeling approaches.

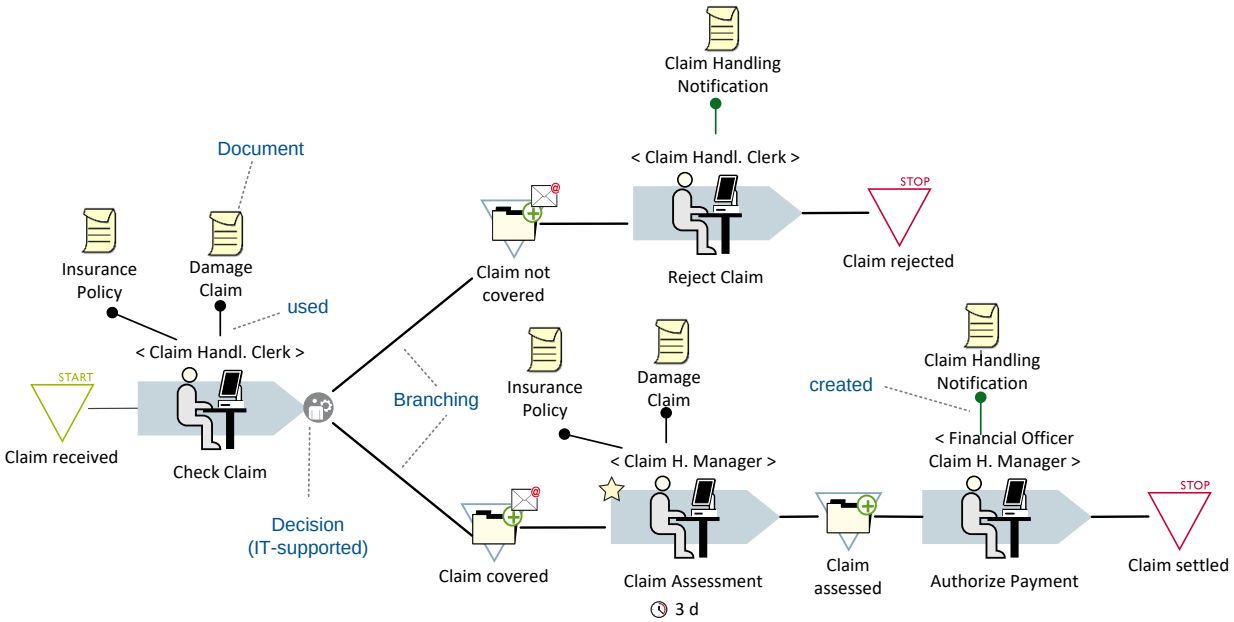

Figure 19: OrgML model of the claims handling process

5.1 Evaluation against Requirements

To check the presented model against the requirements, we will explicitly account for every requirement in the two lists P and S. Tab. 1 presents the results of checking the proposed solution against requirements P1 to P19 related to "processes, tasks, actors, and artifacts". Tab. 2 relates to requirements S1 to S13. If we conclude that a requirement is clearly satisfied, the corresponding evaluation is marked with an "S" if it is not satisfied in a literal sense, we use a "C".

Tab. 3 serves to further facilitate the comparison with regular contributions to the challenge. The entries in the first column describe the additions that go beyond the requirements defined with the challenge.

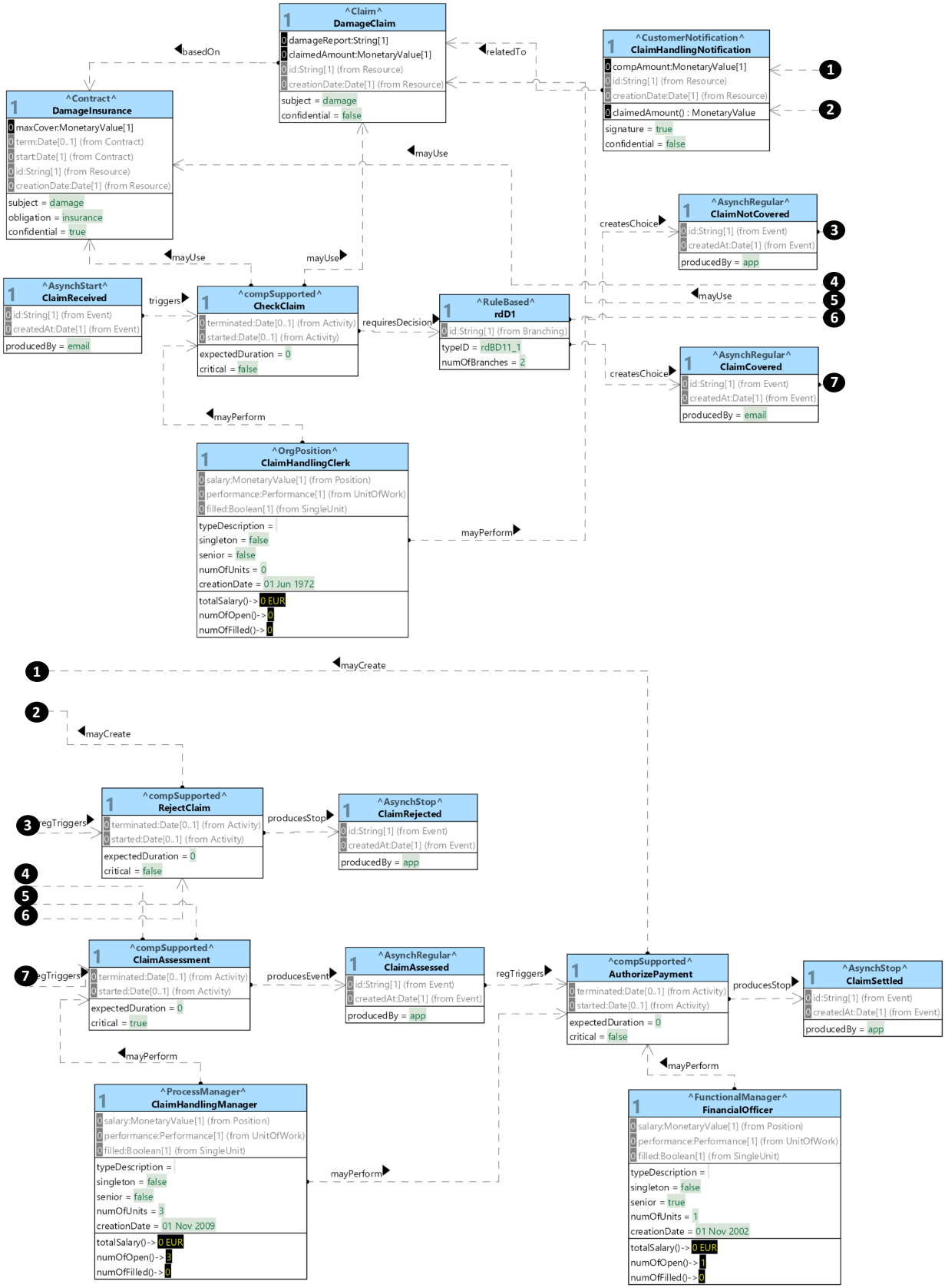

Figure 20: FMML^x model of the claims handling process

Requirement		*Assessment*
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.	**S**	The generic process model in 12 and the models of the software development process and the insurance process in figs. 15, 16 and 20 show that this requirement is clearly satisfied.
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.	**S**	The generic process model defines all control flow elements mentioned in the requirement, with the only difference that other designators are used for naming gateways.
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).	**S**	This requirement is satisfied on the type level by the intrinsic associations **terminatedBy** and **startedBy** between **BusinessProcess** and **StopEvent** and **StartEvent** respectively. In addition, the intrinsic association **beginsWith** is to link the representation of particular business process with the starting event on L0.
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.	**S**	The intrinsic association **designs** between **SingleUnit** and **Activity** allows the realization of this requirements. In addition, the association **defines** between selected position types and **BusinessProcess** allows to specify that the holder of a certain position defines all task types of a certain process types (cf. Fig. 13).
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.	**S**	In the generic process model, the intrinsic association **mayPerform** between the classes **SingleUnit** and **Activity**, both on L3, allows assigning classes representing role or position types on L1 to activity types on L1. The constraint **legitimatePerform** within **Activity** checks whether the assignment of a particular role or position to an activity on L0 is legitimated by a corresponding link between their classes on L1.
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).	**C**	We did not include that in the generic process model, because it violates principles of organizational design to define regulations *ad personam*. Possible assignments can, however, be restricted by the solution to P5. Nevertheless, it would, of course, be possible to define an intrinsic association between **ActUnit** and **Activity**, which would be instantiated on L0, however, at the risk of producing contradictions to the constraints defined with the associations described with the solution for P5.
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.	**C**	The artifact types that can be used and produced by a task type can be defined through instantiations of the intrinsic associations **mayCreate** and **mayUse** defined. However, in our model decisions are not conceptualized as artifacts. Instead, they are represented by branchings and the alternative event types that they produce.
P8: Task types have an expected duration.	**S**	The intrinsic attribute **expectedDuration** of **Activity**, to be instantiated at L1, serves the representation of this property.
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.	**C**	The intrinsic attribute **critical** of **Activity** allows characterizing a task type as critical. The constraint **criticalTask** of **Activity** checks, whether a critical task type is assigned to a position type that qualifies as senior. The requirement that the artifacts produced by a critical activity need to be linked to a validation task is accounted for by the constraint **properValidation** in **Activity**. For reasons outlined in Sect. 4.4.1, the constraint is limited by the lack of further information.
P10: Each process type may be enacted multiple times.	**S**	Process types are represented on L1 and, hence, allow for multiple instantiations.
P11: Each process comprises of one or more tasks.	**S**	Each business process has exactly one start event, which in turn triggers an activity directly or indirectly through a fork.
P12: Each task has a begin date and an end date.	**S**	The intrinsic attributes **started** and **terminated** of **Activity** allow assigning a begin and end date to task at L0.

Continued on next page

Table 1 – *Continued from previous page*

Requirement		*Assessment*
P13: Tasks are associated with artifacts used and produced, along with performing actors.	**S**	Obviously satisfied by various associations between **Activity** on the one hand, and the classes **Resource** and **SingleUnit** on the other hand (cf. fig 12).
P14: Every artifact used or produced in a task must instantiate one of the artifact types stipulated for the task type.	**S**	The various assocations between **Activity** and **Resource**, as well as the constraints **legitimateCreation** and **legitimateUse** of **Activity** satisfy this requirement.
P15: An actor may have more than one actor type	**C**	As we outlined already in Sect. 3, we struggle with this requirement, because the FMML^x (like any other object-oriented language we know) does not allow for multiple classification (multiple generalization is different!). Our model accounts for this requirement by using roles that can be assigned to an employee in addition to her position.
P16: Likewise, an artifact may have more than one artifact type.	**C**	For the same reasons that apply to P15, we cannot satisfy this requirement. Different from P15, we do not see how the use of roles, that is, of delegation, would help. If, however, a relaxed interpretation is applied to the requirement, our solution would be satisfactory: a particular contract, for example, may be an instance of the class **InsurancePolicy** and a descendant of **Document** at the same time.
P17: An actor who performs a task must be authorized for that task.	**S**	Clearly satisfied by the associations between **Activity** and **SingleUnit** and additional constraints; corresponds to P5.
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.	**C**	This requirement is not directly realized by the proposed solution for reasons explained in Sect. 3. Nevertheless, the generic process models allows addressing a slightly modified version of this requirement through the constraint **legitimatePerform** in **Activity**: a superior may perform every task one of her subordinates is authorised to carry out, as long as this would not violate additional constraints like, e. g., that a program may be manipulated only by actors who master the corresponding language.
P19: All modeling elements, at all levels, must have a last updated value of type time stamp.	**S**	This requirement is satisfied by the attribute **lastUpdated** in **MetaClass** (see Fig. 1), since all FMML^x classes inherit from **MetaClass**. A more sophisticated solution would include the automatic update of the corresponding slot value. This could be achieved through a modification of the meta object protocol, which we did not do, since that implies a remarkable effort and would go beyond the requirement defined with the challenge. Note that the attribute and the corresponding slot values are not shown in the diagrams that represent the solution. This is due to the general decision not to show attributes or operations defined in **MetaClass** or in **Class** in diagrams, like, e. g., the attribute **name** or the operation **allInstances()**.

Table 1: Evaluation against requirements P1 to P19

Requirement		*Assessment*
S1: A requirements analysis is performed by an analyst and produces a requirements specification.	**S**	Satisfied by the software development process in Fig. 15.
S2: A test case design is performed by a developer or test designer and produces test cases ... only senior analysts may perform a test case design.	**S**	The first part of the requirement is directly addressed by the software development process model, the second part is also accounted for by the constraint **criticalTask**, which checks whether the type of a position or role assigned to an instance of a critical task type is qualified as senior.
S3: An occurrence of coding is performed by a developer and produces code. It must furthermore reference one or more programming languages employed.	**S**	The instantiation of the intrinsic association **mayCreate** between **SingleUnit** and **Activitiy** in the software development process satisfy the first part. The software artifact model defines that every particular code document needs to refer to the language it was created with.
S4: Code must reference the programming language(s) in which it was written.	**S**	That corresponds to the second part of S3.
S5: Coding in COBOL always produces COBOL code.	**S**	Every particular code document, including COBOL code, needs to be linked to the language in which it was written (cf. S 4). Therefore, the requirement seems to be redundant.
S6: All COBOL code is written in COBOL.	**S**	Again, with respect to the proposed solution, this requirement is redundant.
S7: Ann Smith is a developer; she is the only one allowed to perform coding in COBOL.	**S**	The constraint **allowedToManipulate** defines a corresonding rule that may lead to Ann Smith being the only one who is allowed to perform coding in COBOL if she happens to be the only one who masters COBOL.
S8: Testing is performed by a tester and produces a test report.	**S**	Directly represented by the sofware development process model, see 16.
S9: Each tested artifact must be associated to its test report.	**C**	We are afraid that this requirement cannot be satisfied without further information. While it would be no problem to associate the class **Code** with the class **TestReport** (see Fig. 10), it is not clear, whether this should apply to every code document and what other documents require testing. Furthermore, it is not clear, whether a test report has to be created within the same process the document to be tested was created in.
S10: Software engineering artifacts have a responsible actor and a version number.	**C**	The intrinsic attribute **version** in **SofwareArtifact** allows assigning a version number to every artifact at L0. A responsible actor is assigned via the activity during which an artifact is manipulated. Hence, our solution is based on the assumption that artifacts are manipulated only within processes.
S11: Bob Brown is an analyst and tester. He has created all task types in this software development process.	**S**	Directly represented in the model shown in Fig. 13.
S12: The expected duration of testing is 9 days.	**S**	Represented in the software development process model (see Fig. 16).
S13: Designing test cases is a critical task which must be performed by a senior analyst. Test cases must be validated by a test design review.	**S**	Both aspects of this requirement are accounted for in the model of the software development process. It is a matter of interpretation, whether the assignments made in the software development process are valid for all processes where test cases are designed, which leads to further questions we shall discuss in Sect. 6.

Table 2: Evaluation against requirements S1 to S13

Additional Concepts	*Rationale*
Concepts to describe organizational structures on L 3, e. g., **Position**, **CompOrgUnit**, **Role**. See Fig. 6.	These concepts serve the representation of text-book knowledge on organizational structures. The purpose of this additional abstraction is to provide a DSML that can be used for (almost) any kind of organization. At the same time, it is clearly more restrictive than concepts known from GPMLs, thus contributing to model integrity.
Concepts to describe more specific organizational structures on L2. See Fig. 7.	These classes represent more specific language concepts. Together with more generic concepts they demonstrate that multi-level modeling is suited to mitigate the notorious conflict between range of reuse (high on a more generic level) and productivity of reuse (growing with lower levels). The lower levels benefit from reusing concepts defined on higher levels.
Introduction of classes to represent specific positions and roles instead of using a more general concept like actor.	Similar to the introduction of high level concepts for describing organizational structures, these concepts represent reusable domain-specific knowledge that supports the convenient and safe creation of artifact types that are characteristic for a specific domain.
Classes on L3 and L2 to represent process modeling concepts.	While the intention here is similar to the previous one, the effect of this measure is less convincing. For reasons outlined in the paper, higher level concepts for process modeling foster reuse of specific static or functional aspects of process models on lower levels, but they do not allow for reusing dynamic aspects such as, e. g., a certain constraint on a sequence of activities that would apply to process models on lower levels, too.
Introduction of specific operations like **totalSalary()** in **Position** or **claimedAmount()** in **ClaimHandlingNotification**.	At first, these operation serve to demonstrate that the presented models are executable. In addition, they demonstrate the utility of intrinsic operations.
Introduction of additional constraints like **properPart0** or **properPar1** of **CompOrgUnit**, or **allowedToManipulate** of **Activity**.	Adding constraints aims to demonstrate the utility of active constraints enabled by the XModeler^{ML} . Whenever a change operation on a model violates a constraint, an alert is created an shown within the diagram. Furthermore, intrinsic constraints show how to restrict more specific DSML through knowledge that is available with more generic concepts already.
Use of delegation to specify the relationship between the classes **Person** and **Employee**.	This extension is motivated by the need to allow for more elaborate specifications of organizational responsibilities than those possible through a generic concept like actor. Also, it is to demonstrate that the XModeler^{ML} implements delegation and, thus, allows to transparently access the state of role filler objects by role objects.

Table 3: Elements of the solution that go beyond the challenge

5.2 Comparison with Traditional Language Architectures

During the last 30 years, one of the authors was involved in the development of numerous modeling languages and corresponding modeling tools, which were designed within traditional language architectures that allow one classification level only, for example, (Frank 1994, 1998, 2011a,b; Frank et al. 2009; Gulden and Frank 2010; Kirchner 2005; Overbeek et al. 2015). Against this background we shall at first perform a general comparison of multi-level languages against languages that follow the traditional paradigm. Subsequently, we look at specific features offered by the FMML^{x} and the XModeler^{ML} . For a comprehensive analysis of specific benefits of multi-level modeling and its comparison to traditional approaches see (Frank 2022).

5.2.1 General Assessment of Multi-Level Languages

The design of modeling languages aims at providing prospective users with an instrument that facilitates the convenient creation of consistent models. In the case of DSML, the models are restricted to certain domains. The convenience of modeling corresponds to productivity and to ease of use. While both aspects are not independent from the concrete syntax, we fade this aspect out, because both, traditional language and multi-level languages may benefit from a thoroughly designed notation alike.

Among the limitations of languages of the traditional paradigm, the lack of expressiveness is probably the most frustrating one. Whenever you build a system you would like to express the knowledge you have at the appropriate level of abstraction. For example, a metamodel can be used to describe properties of a position type, but even though we know the specific characteristics that apply to particular positions only (salary, filled by a certain employee, etc.), we cannot express them. If that is not possible you are forced to develop workarounds that are likely to increase a system's complexity and, as a consequence, are a threat to system integrity. Unfortunately, with the design of DSML this is not a rare exception, but a rule. Multi-level modeling overcomes limitations of expressiveness that are inherent in the old paradigm. The models presented in this paper include numerous examples. All intrinsic features used in these models could not have been expressed in the traditional paradigm.

A further source of great frustration with the old paradigm is the fact that the specification of a language always starts from scratch. Meta-modeling languages and meta-modeling tools offer basic language concepts such as "class", "attribute", etc. only. This is a remarkable restriction. Imagine, we would have to write this paper with a primitive language like that! As a consequence, the design and implementation of languages is costly and error-prone. Another problem is related to the previous one: the need to clearly distinguish between language and language application. Unfortunately, there are no clear rules to make this distinction. For example, one has to decide whether notions like "organizational unit" or "department" should be modelled as language concepts or rather be specified with the language. If one decides for organizational unit, department would not qualify as language concept anymore, even though it is definitely part of technical languages in the domain of organizational design. A multi-level model does not require this distinction. Both notions can be included in one language, on different levels of abstraction.

In addition, and related to the previous, the design of languages within the traditional paradigm is confronted with a sometimes frustrating decision. On the one hand, the utility of a language depends on its reach. The more people use it, the better are economies of scale. On the other hand, its utility depends on its specificity. The better it is tailored to a particular domain, the higher is its contribution to productivity and model integrity. For all the DSML that we developed within the traditional paradigm we were forced to decide for a particular trade-off between economies of scale and modeling productivity — even though both are of pivotal relevance. As the design of the multi-level models presented in this paper clearly

illustrates, that conflict can be substantially relaxed. The generic models enable a wide range of reuse and the more specific models, while benefiting from reusing the concepts of a more general DSML, allow the adaptation to specific needs. This is a huge economic advantage enabled by the additional abstraction provided by multi-level languages.

Despite these obvious benefits, it may seem at first sight that multi-level modeling suffers from a serious drawback. The solution we present in this paper is of remarkable complexity. It is indeed the case that the higher level of a multi-level model may be characterized by specific kinds of complexity not known of in the traditional paradigm. The generic process model and the generic model of organizational structures are obvious examples. However, even though this objection cannot be ignored, it is hardly convincing in the end. First, it would not be less complex to satisfy the requirements of the challenge with traditional language concepts than with the intrinsic features and especially the constraints specified in the generic models. To the contrary, it would be necessary to overload concepts and to add more complex constraints to somehow create models of comparable semantics. Second, the degree of complexity inherent to a multi-level model varies. The higher the level of abstraction, the more complex and demanding the construction (and understanding) of a multi-level model gets. At the same time, convenience and integrity of use increase on lower levels, because the concepts defined on higher levels already enable reuse and guidance. This corresponds to the general rule that the reduction of complexity implies to first increase it.

5.2.2 Specific Features of the FMML^{x} and the XModeler^{ML}

In addition to multi-level modeling in general, the FMML^{x} and its implementation with the XModeler^{ML} offer specific features that had an impact on the proposed solution. First, the ability to define associations across different levels proved to be very useful. Otherwise, it would not have been possible expressing, e. g., that an employee can fill any kind of position at a level where particular position types were still abstracted away. Also, the definition of separate instantiation levels within one intrinsic association turned out to be useful, for example, to express that the holder of a certain position may define a business process type (cf. fig 13).

With respect to the language engineering environment provided by the XModeler^{ML}, it is a substantial advantage that all models are executable. The models can be seen as applications that allow for interaction. The state of a model can be changed, either within the diagram editor or by using the model browser. Operations do not only serve the support of specific inquiries, such as computing the total amount of salaries assigned to all instances of a certain position type. In addition, it is possible to add specific control classes that could orchestrate more complex application scenarios, like the enactment of a particular process including the required user interactions.

Models and programs share the same internal representation. Therefore, synchronization of the two is not an issue. Every user is free to use the representation of his choice. A diagram view will often be preferable when the relationships between classes need to be accounted for. The browser is likely to be seen as more convenient, when the focus is on editing operations or constraints. In addition to these standard views, it is possible to create further views in order to build a customized user interface.

6 Discussion

The challenge is mainly motivated by the idea to compare and assess existing approaches to multi-level modeling. In addition, the scenarios presented with the challenge should also serve as a laboratory for evaluating and further developing existing languages and tools. With this in mind, we will first discuss the experiences we had with the FMML^{x} and the XModeler^{ML} during the design of the solution. Then we will point out a key research challenge whose importance is underlined by the process challenge.

6.1 Limitations of Multi-Level Modeling and the FMML^x

The evaluation of the models we propose shows that they satisfy directly most of the requirements defined by the challenge. The remaining requirements are also addressed, but in some cases based on an interpretation that may be slightly different from what the authors of the challenge intended. Therefore, the FMML^x and the XModeler^{ML} served us as effective tools to develop a solution. Nevertheless, we experienced a few limitations of the language and the tool as well. The main drawback we experience with the FMML^x was the insufficient support for representing contingencies. The design of multi-level models recommends expressing the knowledge we have on the highest possible level. It sometimes happens that we have relevant knowledge we would like to express with a class at a certain level, but we cannot exclude, or we may even know that there are exceptions within the concretization tree of that class. Of course, one could cope with this kind of contingency by treating the different cases separately. That would, however, lead to conceptual redundancy: common properties of the affected classes would have to be modelled multiple times. A prominent example are cases where the level of a class may differ with the context, hence, its level is contingent (cf. the brief outline in Sect. 2). The latest extension of the language allows for defining the level of a class as contingent, which means that it can be adapted with the context the class is used in. This feature is, however, not implemented in the current version of the XModeler^{ML} .

While we did not need contingent level classes (or "level jumps"), a modification of the model would have likely created such a demand. If, for example, we had decided to model `SoftwareArtifact` as concretization of `Document`, `Document` would have been at L4, while it is at L3 in the artifact model of the insurance domain. With respect to the obvious differences between both domains that does not seem as a serious drawback. In contrast, we experienced the lack of language concepts to specify multiplicities within intrinsic associations as contingent as a more serious limitation. We could not express that the cardinality of an intrinsic association between `Resource` and `Activity` needs to be changed within some concretizations (see corresponding discussion in Sect. 4.4.3). Both specific limitations indicate that there is need for concepts that enable underspecification in order to cover a wider range of possible concretizations.

The design of the presented models was guided by a preliminary method that goes beyond the principles described in Sect. 4.1. Even though it provides very useful support, the design of the models confirmed two conjectures we made already earlier. First, there is need for extending the method with more specific guidelines that concern, e. g., decisions on the appropriate level of classes or on the use of contingent level classes (for a corresponding proposal see Frank and Töpel 2020). Second, the variety and contingency of domains to be covered by a method implies that a method cannot be comprehensive. Therefore, modelers need to be trained and reflective to make informed and convincing decisions. While this is the case for any modeling method, this insight has, according to our experience, even more weight for multi-level modeling. Note, however, that this does not mean that multi-level modeling is more demanding for all stakeholders. Those who use existing higher level models as languages to create more specific models benefit from less complexity and more guidance, which is clearly demonstrated by the lower level models shown, e. g., in figures 8, 11, 10, or 15. Therefore, the occasional reservation against multi-level modeling that it would be too complicated for most users, is not appropriate. Instead, the development of the presented models once again confirms a general principle: the reduction of complexity implies increasing it at first. This does not come as a surprise. Usually, the development of a DSML is more demanding, i. e., of higher complexity, than using the language.

The XModeler^{ML} proved to be an effective tool to create, and execute, multi-level models. Especially, the option to switch between a diagram and

a browser/editor view was useful in many situations. However, this assessment is not free of bias, because we were involved in the development of the tool. In any case, it is extremely useful for a tool to support model changes, because these can hardly be avoided. The XModeler^{ML} enables deleting and modifying almost all properties of classes, including names, and levels of attributes, associations, and operations. Furthermore, types of attributes can be changed, as well as the properties of associations and the implementation of operations and constraints. It is also possible to lift or lower the level of *all* objects of a model. The latter, however, is usually not required. Instead, we occasionally experienced the need for changing the level of a particular class, which is currently not supported by the XModeler^{ML} . While this kind of change will usually not allow for being fully automated, it would be very useful if the tool guided the user interactively through the required changes.

During the design of the models it became apparent that a tool should provide support for dealing with large models. The model for the software development firm comprises of 84 classes in total, with 17 classes on L3, 30 on L2, and 37 on L1. The model can easily grow to a substantially larger size with further concretizations. The XModeler^{ML} provides two features to cope with the size of diagrams. First, it allows hiding parts of a diagram, while the affected objects are still accessible through the object browser. Second, it is possible to distribute a diagram to different views, which are accessible through tabs at the top of the canvas (see screenshot in Fig. 2). Each view shows a certain part of a model. Views may overlap, that is, parts of a model can be shown in more than one view.

6.2 The Challenge Beyond the Challenge: Multi-Level Modeling of Behavior

The solution we present aims at the requirements defined by the challenge. Therefore, it does not address a further aspect of multi-level process models that is especially relevant and, at the same time, extremely challenging. The solution comprises of static and, to a small degree, functional abstractions only. However, modeling processes suggests to account for dynamic abstractions, too. The state of the art in process modeling is characterized by a remarkable lack of abstraction and, as a consequence, poor reuse and little support of model integrity. The design of a process model always starts from scratch with primitive concepts such as event, activity, etc. In the field of business process modeling, this limitation has been known for long. A popular approach to mitigate it is the development of reference process models (see, e. g., Fettke et al. 2006, Frank and Lange 2004). A thoroughly designed model, for example of an order management process, should serve as a reference for an entire industry or parts of it. Even though the idea is appealing it did not turn into the expected success story. One of the obstacles that hinders the distribution of reference models is their limited adaptability. Reuse is mainly restricted to copy&paste. Various approaches aimed at addressing the resulting maintenance problem such as the conceptualization of process variants (Hallerbach et al. 2010) or the use of rules to constrain the range of change operations (Popp and Kaindl 2015).

Other approaches focus at the core of the problem, the lack of abstraction, by proposing concepts of process specialization. Van der Aalst and Basten use petri nets to define specialization rules for processes (Aalst and Basten 2002) or operations in object-oriented software systems (Schrefl and Stumptner 2002). Wyner and Lee propose an approach to process specialization, which, however, rather qualifies as process instantiation (Wyner and Lee 2002). While these approaches promise some relief, their effect on reuse and adaptability remains modest. This is due to the principle problem that process models do not allow for monotonic extensions of the control flow. As a consequence, the effect of changes applied to a superprocess on its subprocesses is not deterministic (for an elaborate analysis of this problem see Frank 2012).

Applying the idea of multi-level modeling to process models may help to find a way out of this predicament. The following scenario illustrates the idea. At a higher level, a general model of claims handling processes would represent the knowledge that is common to all claims handling processes in the insurance industry. It would, e. g., include that every claim has to be formally checked before it is further analyzed. This model would form a language to describe more specific claims handling processes, e. g., in the field of vehicle insurance. The corresponding language could then be (re-) used to define a more specific process model, for a certain type of insurance policy or for a certain company. While we believe that this is an appealing idea that has the potential to clearly improve the current state of the art of process modeling, we are aware of the challenges it brings. This assessment was confirmed by a recent Dagstuhl seminar on multi-level modeling (João Paulo A. Almeida et al. 2018). While most of the participants regarded multi-level process modeling as a formidable challenge, they also agreed that multi-level modeling is suited to improve reuse and adaptability of process modeling languages and process models alike.

7 Related Work

The models proposed as a solution to the challenge build in part on previous work on traditional, non-multi-level DSMLs. The MEMO OrgML, for designing both organizational structures (Frank 2011a) and business process models (Frank 2011b), makes use of similar abstractions, but suffers from the limitations of traditional language architectures. That is the case, too, for the comprehensive environment for enterprise modeling (Frank and Bock 2020) that implements the language.

There are three papers that directly relate to the work presented here. In his contribution to process challenge, presented at MULTI 2019, Jeusfeld proposes a multi-level model designed with DeepTelos (Jeusfeld 2019). DeepTelos is based on first-order logic. The language is specified as a model of a theory defined by 30 axioms (Jeusfeld 2021). Its semantics is therefore different from the FMML^{x} and other multi-level languages that are based on object-oriented concepts. In particular, an object in DeepTelos can be of more than one class. The solution presented to the MULTI 2019 challenge builds on a specification of the BMPN-core with Telos, the predecessor of DeepTelos (Jeusfeld 2021, p. 8). Compared to our solution, it puts less emphasis on modeling the organizational context and resources. Therefore it does not include aspects of organizational governance, which we accounted for in some detail. In part, the proposed models are based on interpretations of the requirements that are different from ours. For example, “CodingCobol” is modelled as a subclass of the task type “Coding”, which is linked to the object representing Ann Smith, thus representing a “lex Smith” that we wanted to avoid.

Somogyi et al. present a solution that is based on an idiosyncratic approach to multi-level modeling called “Dynamic Multi-Layer Algebra” (Somogyi et al. 2019). While a detailed discussion of the proposed models would go beyond the scope of this paper, it is noticeable that some of the models are challenging to read. For example, the class *SeniorAnalyst* is instantiated from *SeniorActorType* and specialized into *AnalystJoe*, which in turn is an instance of *Actor*. The contribution by Rodriguez and Macias (Rodriguez and Macias 2019), which is based on MultEcore presents a solution with an overall structure similar to ours. However, it lacks an explicit representation of organizational structures. Also, the process model is less elaborate, since it does not include events.

Kaczmarek-Heß and de Kinderen developed a multi-level model of organizational structures (Kinderen and Kaczmarek-Heß 2020). It was designed with the FMML^{x} and an earlier version of the XModeler^{ML}. Similar to our solution, it builds in part on the OrgML. The classes `OrganizationalUnit` and `Position` are also located at level 3. Classes are modelled in more detail, that is, they comprise of more attributes. Also, the

model includes a class to represent organizations, which allows to account for various organizations/companies in one model. The model does not make use of the composite pattern to represent organizational structures and does not introduce a class dedicated to the representation of employees.

The "Organization Ontology" published by the W3C (W3C 2014), is a recommendation for the representation of organizational structures. The model is specified with RDF. Its main purpose is "supporting linked data publishing of organizational information across a number of domains". To that end, the ontology remains on a high, rather superficial level. The documentation of the ontology includes illustrations of how concepts can be refined to adapt to specific organizations, e. g. by adding "qualifying information" like attributes or by decomposing concepts, like "corporation" into "business unit", "division", etc. Therefore, the organization ontology is clearly less elaborate than the generic model of organizational structures in this paper. It is, however, aimed at a different purpose, which restricts comparability of the two models.

Various approaches exist that aim at the specification of process modeling languages, such as EPC (Keller et al. 1992), BMPN (OMG 2013) or UML activity diagrams (OMG 2015) in general, or workflow management languages, e. g., BPEL (OASIS 2007) or YAWL (van der Aalst and ter Hofstede 2005) in particular. With respect to the purpose of these specifications, that is, the definition of general (business) process modeling languages, it would be beside the point to compare them against our process model the design of which was aimed at the requirements defined in the challenge. Nevertheless, already a brief look at the BMPN specification gives an impression of obvious limitations caused by a traditional language architecture. While the BMPN, for obvious reasons, abstracts domain-specific characteritics of business processes away, it nevertheless needs to provide some kind of adaptation mechanisms. This is mainly achieved by allowing users to add further attributes (OMG 2013, p. 57). It is not possible, however, to define new language concepts, as it is the case for multi-level models. While resources are mandatory for the execution of business processes, the specification of resources in BPMN remains shallow, again for the reason that the variety of resource types is enormous. The multi-level resource models that we present indicate how variety and the need for adaptability can be coped with.

8 Conclusions

The design of the models presented in this contribution was related to an effort that was clearly higher than we expected. That was mainly due to the fact that we had to change models several times, even though we had prepared the design thoroughly. We believe that this effort paid off. First, it helped us to further reflect upon principles and obstacles of developing multi-level models. Second, the result should be suited to demonstrate clear benefits enabled by the additional abstraction offered by multi-level models. We hope that the models presented in this paper contribute to a critical discussion that compares specific aspects of the different contributions to the challenge and the corresponding tools, since further research on multi-level modeling depends on competition. At the same time, we will appreciate it if competition is complemented by a concerted effort to consolidate the research field, and to regard the development of multi-level models of behavior as a community challenge. If we succeed in bundling resources, appreciation and dissemination of multi-level modeling, which, after all, represents a major progress in conceptual modeling, should be clearly strengthened.

The latest version of the XModeler^{ML} is available in the download section of the web pages of the project LE4MM ("Language Engineering for Multi-Level Modeling"). The models presented in this paper can be downloaded from there, too. Corresponding screencasts demonstrate the installation of the XModeler^{ML} , as well as loading and using the provided models (https://www.wi-inf.uni-duisburg-essen.de/LE4MM/downloads/).

References

van der Aalst W. M. P., Basten T. (2002) Inheritance of Workflows: An Approach to Tackling Problems Related to Change. In: Theoretical Computer Science 270(1-2), pp. 125–203

Almeida J. P. A., Rutle A., Wimmer M., Kuhne T. (2019) The MULTI Process Challenge. In: Staff I. (ed.) 2019 ACM IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS C). IEEE, pp. 164–167

Atkinson C., Kühne T. (2001) The Essence of Multilevel Metamodeling. In: Gorgolla M., Kobryn C. (eds.) UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Lecture Notes in Computer Science. Springer, pp. 19–33

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

Balaban M., Khitron I., Kifer M., Maraee A. (2018) Multilevel Modeling: What's in a Level? A Position Paper. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 693–697

Clark T., Sammut P., Willans J. S. (2015a) Applied Metamodelling: A Foundation for Language Driven Development (Third Edition). In: CoRR abs/1505.00149

Clark T., Sammut P., Willans J. S. (2015b) Super-Languages: Developing Languages and Applications with XMF (Second Edition). In: CoRR abs/1506.03363

Fettke P., Loos P., Zwicker J. (2006) Business Process Reference Models: Survey and Classification. In: Bussler C. J., Haller A. (eds.) Business Process Management Workshops. Lecture Notes in Computer Science Vol. 3812. Springer, pp. 469–483

Frank U. (1994) Multiperspektivische Unternehmensmodellierung: Theoretischer Hintergrund und Entwurf einer objektorientierten Entwicklungsumgebung. Oldenbourg

Frank U. (1998) The MEMO Object Modelling Language (MEMO-OML)

Frank U. (2000) Delegation: An Important Concept for the Appropriate Design of Object Models. In: Journal of Object-Oriented Programming 13(3), pp. 13–18

Frank U. (2011a) MEMO Organisation Modelling Language (1): Focus on Organisational Structure

Frank U. (2011b) MEMO Organisation Modelling Language (2): Focus on Business Processes

Frank U. (2011c) The MEMO Meta Modelling Language (MML) and Language Architecture. 2nd Edition. http://www.icb.uni-due.de/fileadmin/ICB/research/research_reports/ICB-Report_No43.pdf

Frank U. (2012) Specialisation in Business Process Modelling: Motivation, Approaches and Limitations

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

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

Frank U. (2016) Designing Models and Systems to Support IT Management: A Case for Multilevel Modeling. In: Proceedings of MULTI 2016. CEUR-WS.org, pp. 3–24

Frank U. (2021) Prolegomena of a Multi-Level Modeling Method Illustrated with the FMML^{x}. In: Proceedings of the 24th ACM/IEEE International Conference on Modell Driven Engineering Languages and Systems: Companion Proceedings. IEEE

Frank U. (2022) Multi-level modeling: cornerstones of a rationale. In: Software and Systems Modeling (Online First)

Frank U., Bock A. (2020) Conjoint Analysis and Design of Business and IT: The Case for Multi-Perspective Enterprise Modeling. In: Kulkarni V., Reddy S., Clark T., Barn B. (eds.) Advanced Digital Architectures for Model-Driven Adaptive Enterprises. IGI Global, pp. 15–45

Frank U., Heise D., Kattenstroth H., Ferguson D., Hadar E., Waschke M. (2009) ITML: A Domain-Specific Modeling Language for Supporting Business Driven IT Management. In: Rossi M., Gray J., Sprinkle J., Tolvanen J.-P. (eds.) Proceedings of the 9th OOPSLA workshop on domain-specific modeling (DSM'09). Helsinki Business School

Frank U., Lange C. (2004) A Framework to Support the Analysis of Strategic Options for Electronic Commerce

Frank U., Töpel D. (2020) Contingent Level Classes: Motivation, Conceptualization, Modeling Guidelines, and Implications for Model Management. In: Guerra E., Iovino L. (eds.) Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings. ACM, pp. 622–631

Guerra E., Lara J. D. (2018) On the Quest for Flexible Modelling. In: Wąsowski A., Engineering A. S. I. G. o. S. (eds.) Proceedings of the 21th ACMIEEE International Conference on Model Driven Engineering Languages and Systems. ACM, pp. 23–33

Gulden J., Frank U. (2010) MEMOCenterNG – A full-featured modeling environment for organisation modeling and model-driven software development. In: Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development (FTMDD 2010)

Hallerbach A., Bauer T., Reichert M. (2010) Configuration and Management of Process Variants. In: Vom Brocke J., Rosemann M. (eds.) Introduction, methods and information systems. Handbook on business process management. Springer, pp. 237–255

Igamberdiev M., Grossmann G., Stumptner M. (2016) A Feature-based Categorization of Multi-Level Modeling Approaches and Tools. In: Atkinson C., Grossmann G., Clark T. (eds.) Proceedings of the 3rd International Workshop on Multi-Level Modelling (MULTI 2016), Saint-Malo, France. CEUR Workshop Proceedings Vol. 1722. CEUR-WS.org, pp. 45–55

Jácome-Guerrero S. P., de Lara J. (2020) *TOTEM*: Reconciling Multi-Level Modelling with Standard Two-Level Modelling. In: Computer Standards & Interfaces 69, p. 103390

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

Jeusfeld M. (2019) DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 66–77

Jeusfeld M. (2021) ConceptBase.cc User Manual: Version 8.1. http://conceptbase.sourceforge.net/userManual81/

João Paulo A. Almeida, Ulrich Frank, Thomas Kühne (2018) Multi-Level Modelling (Dagstuhl Seminar 17492). In: Dagstuhl Reports 7(12), pp. 18–49

Kaczmarek-Heß M., de Kinderen S. (2017) A Multilevel Model of IT Platforms for the Needs of Enterprise IT Landscape Analyses. In: Business & Information Systems Engineering 59(5), pp. 315–329

Keller G., Nüttgens M., Scheer A.-W. (1992) Semantische Prozeßmodellierung auf der Grundlage "Ereignisgesteuerter Prozeßketten (EPK)". http://www.uni-saarland.de/fileadmin/user_upload/Fachrichtungen/fr13_BWL/professuren/PDF/heft89.pdf

de Kinderen S., Kaczmarek-Heß M. (2020) On Model-Based Analysis of Organizational Structures: an Assessment of Current Modeling Approaches and Application of Multi-Level Modeling in Support of Design and Analysis of Organizational Structures. In: Software and Systems Modeling 19(2), pp. 313–343

Kirchner L. (2005) Cost Oriented Modelling of IT-Landscapes: Generic Language Concepts of a Domain Specific Language. In: Desel J., Frank U. (eds.) Enterprise Modelling and Information Systems Architectures. Lecture Notes in Informatics. Gesellschaft für Informatik, pp. 166–179

Kühne T. (2018) A Story of Levels. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 673–682

Kühne T., Schreiber D. (2007) Can programming be liberated from the two-level style: multi-level programming with deepjava. In: Gabriel R. P., Bacon D. F., Lopes C. V., Steele G. L. (eds.) Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA 2007). ACM SIGPLAN notices. ACM, pp. 229–244

de Lara J., Guerra E. (2010) Deep Meta-modelling with MetaDepth. In: Vitek J. (ed.) Objects, Models, Components, Patterns. Springer, pp. 1–20

de Lara J., Guerra E., Cuadrado J. S. (Dec. 2014) When and How to Use Multilevel Modelling. In: ACM TOSEM 24(2), 12:1–12:46

Macías F., Rutle A., Stolz V., Rodríguez-Echeverría R., Wolter U. (2018) An Approach to Flexible Multilevel Modelling. In: Enterprise Modelling and Information Systems Architectures 13, pp. 1–35

Mezei G., Theisz Z., Urban D., Bacsi S. (2018) The bicycle challenge in DMLA, where validation means correct modeling. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops, co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 643–652

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. APCCM '09. Australian Computer Society, Inc., Wellington, New Zealand, pp. 107–116

Neumayr B., Schrefl M. (2009) Multi-Level Conceptual Modeling and OWL. In: Heuser C. A., Pernul G. (eds.) Advances in Conceptual Modeling - Challenging Perspectives. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, pp. 189–199

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

OASIS (2007) Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

OMG (2013) Business Process Model and Notation (BPMN): Version 2.0.2. http://www.omg.org/spec/BPMN/2.0

OMG (2015) OMG Unified Modeling Language: Version 2.5. http://www.omg.org/spec/UML/2.5/PDF

Overbeek S., Frank U., Köhling C. A. (2015) A Language for Multi-Perspective Goal Modelling: Challenges, Requirements and Solutions. In: Computer Standards & Interfaces 38, pp. 1–16

Popp R., Kaindl H. (2015) Automated refinement of business processes through model transformations specifying business rules. In: Rolland C. (ed.) 2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS 2015). IEEE, pp. 327–333

Rodriguez A., Macias F. (2019) Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 152–163

Rossini A., de Lara J., Guerra E., Nikolov N. (2015) A Comparison of Two-Level and Multi-level Modelling for Cloud-Based Applications. In: Taentzer G., Bordeleau F. (eds.) Modelling Foundations and Applications: Proceedings of the 11th European Conference, ECMFA 2015, held as Part of STAF 2015, L'Aquila, Italy. Springer, pp. 18–32

Schrefl M., Stumptner M. (2002) Behavior-consistent specialization of object life cycles. In: ACM Trans. Softw. Eng. Methodol. 11(1), pp. 92–148

Selway M., Stumptner M., Mayer W., Jordan A., Grossmann G., Schrefl M. (2017) A Conceptual Framework for Large-Scale Ecosystem Interoperability and Industrial Product Lifecycles. In: Data & Knowledge Engineering 109, pp. 85–111

Somogyi F. A., Mezei G., Urban D., Theisz Z., Bacsi S., Palatinszky D. (2019) Multi-level Modeling with DMLA - A Contribution to the MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 119–127

Tony Clark, Ulrich Frank (2018) Multi-Level Constraints. In: Regina Hebig, Thorsten Berger (eds.) Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 103–117

van der Aalst W., ter Hofstede A. (2005) YAWL: yet another workflow language. In: Information Systems 30(4), pp. 245–275

Volz B. W. (2011) Werkzeugunterstützung für methodenneutrale Metamodellierung

W3C (2014) The Organization Ontology. https://www.w3.org/TR/2014/REC-vocab-org-20140116/

Wyner G. M., Lee J. (2002) Process Specialization: Defining Specialization for State Diagrams. In: Computational & Mathematical Organization Theory 8(2), pp. 133–155