Multi-Level Modelling with MultEcore

A Contribution to the Multi-Level Process Challenge

Alejandro Rodríguez*,[a], Fernando Macías[b]

[a] Western Norway University of Applied Sciences, Bergen, Norway
[b] IMDEA Software Institute, Madrid, Spain

Abstract. *The MULTI Challenges are intended to encourage the Multi-Level Modelling research community to submit solutions to the same, well described problem. This paper presents one solution in the context of process management, where universal properties of process types along with task, artefact and actor types, together with possible particular occurrences for scoped domains, are modelled. We discuss our solution, detailing how we handle each requirement and explain how we use the different features that the MultEcore tool supports to construct the proposed process case study. We not only focus on the structural dimension of the proposed system where the different models that define the language are provided, but also explore the specification of the static semantics to verify structural constraints and dynamic semantics that refer to the behavioural aspect of the modelled system.*

Keywords. Multi-Level Modelling • Semantics • Model Transformations • MultEcore

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

1 Introduction

Research in Multi-Level Modelling (MLM) is continuously increasing and MLM approaches and tools are getting more mature and varied. The MULTI challenges were created to enhance discussion and facilitate contributions within the MLM community. Encouraging researchers to submit solutions to a common challenge makes it possible to compare them and fosters improvements towards the same set of common goals. In this paper, we use the MultEcore tool (Macías et al. 2016) to create models by applying various multi-level constructions, which are key to fulfilling the criteria established for the MULTI Process challenge (Almeida et al. 2021, 2019). MultEcore enables multi-level modelling through the Eclipse Modelling Framework (EMF) (Steinberg et al. 2008), and therefore allows reusing the existing EMF tools and plugins. The MultEcore tool is available on its webpage[1] and the Eclipse projects which contain all the artefacts of our solution to this challenge can be downloaded from a GitHub repository.[2]

With MultEcore, modellers can create flexible multi-level structures of models that can in turn be composed with each other to include additional aspects. This process is mainly done by defining multi-level hierarchies, where usually the main one is called the *application hierarchy* and the additional ones are called *supplementary hierarchies*. Using parallelism to model Software

* Corresponding author.
E-mail. arte@hvl.no

This work has been partially supported by Comunidad de Madrid as part of the project 49/520608.9/18 (MADRIDF-LIGHTONCHIP) co-funded by ERDF Funds of the European Union.

Note: This work is based on and extends our previous publication to the MULTI 2019 workshop: *Multi-Level Modelling with MultEcore: A contribution to the MULTI Process challenge*.

[1] MultEcore website: https://ict.hvl.no/multecore/
[2] GitHub repository with all the artefacts of the MultEcore solution: https://github.com/MultEcore/no.hvl.multecore.examples.emisa.process2021

Product Lines, an application hierarchy could be understood as the base language module in the context of a language product line (Méndez-Acuña et al. 2016). Supplementary hierarchies can therefore be used to add new dimensions to the application one, with concepts that are not part of the latter's domain. An application hierarchy can include several supplementary hierarchies which can also be removed without introducing inconsistencies or affecting the integrity of the models.

We also take advantage in this paper of the newest features in MultEcore, some of which are part of our current development efforts. In particular, we discuss the specification of constraints (static semantics) and behaviour (dynamic semantics) of the models by applying MultEcore's model transformation language, which we call Multilevel Coupled Model Transformations (MCMTs) (Macías 2019; Macías et al. 2019; Rodríguez et al. 2019a). The key aspects of our framework which have been applied to solve the challenge are summarised as follows:

- The definition of multi-level hierarchies in a flexible way has allowed us to create tree-like structures where the commonalities of the language are defined once, and the branches can be separately specified and instantiated in a controlled manner by using the notion of *potency*, which restricts the levels at which an element can be used to type other elements. Moreover, we use our three-valued definition of potency, which is able to unify consistently the kinds of potency defined, among others, in Atkinson and Gerbig (2016) and Lara and Guerra (2010).
- Being able to define supplementary hierarchies helped us with one of the requirements of the challenge (time-stamping nodes).
- The combination of inheritance (i. e. specialisation) and typing (i. e. instantiation) relations, which can be used together consistently in our approach, has been exploited to address certain requirements.
- We benefited from the two main applications of MCMTs to both check the structural correctness of the modelled hierarchies and to specify behavioural descriptions of the bottom-most models.
- An infrastructure that connects MultEcore with the maude system (Clavel et al. 2007) allowed us to execute our models applying the behavioural MCMT rules.

In this edition, the challenge concerns the domain of process management, which includes both the particular instantiations of elements (e. g., *process instances*, *task occurrences*), and the universal aspects of the domain (e. g. *process definitions*, *task types*). Note that we use British English throughout the paper, however, we use the original US English of the challenge description for quotations and for the names of our elements in the models. For example the reader may find *artifact* being used instead of *artefact* in some contexts. Respondents to the challenge are required to define, first, universal concepts for process management, and second, an application of such a conceptualisation in the scope of a particular software engineering process. Optionally, they can also capture a different scope for the insurance domain. In order to demonstrate the flexibility of our framework, we have defined a multi-level hierarchy where both domains are included.

The rest of this paper is organised according to the structure recommended in the challenge description, as follows. We describe in Sect. 2 the technological aspects of MultEcore. In Sect. 3, we analyse the challenge description and clarify all the assumptions and decisions that we have made in order to fulfil the proposed requirements. We detail in Sect. 4 all the specific elements which are contained in the multi-level hierarchies of our solution, presenting both the software engineering and the insurance domains. We also discuss how we handle cross-level constraints and the operational semantics. In Sect. 5, we discuss each requirement and describe how we addressed it in our solution. In Sect. 6, we assess the choices we have made and describe how certain aspects of our approach facilitate the resolution of the requirements. We discuss in Sect. 7 related work with respect to other solutions developed for past editions of the challenge, both in terms of the approaches used and the solutions developed. Finally, we summarise and conclude the paper in Sect. 8.

2 Technology

Our solution has been entirely modelled using the MultEcore tool (Macías et al. 2017; Rodríguez and Macías 2019), formally specified in Macías (2019) and Wolter et al. (2019). The MultEcore tool is designed as a set of Eclipse plugins, giving access to its mature tool ecosystem (through integration with EMF) and incorporating the flexibility of MLM. In the MultEcore approach (Macías et al. 2019), the abstract syntax is provided by a set of models that compose the language. The semantics in MultEcore (behaviour and constraints) can be specified by using Multilevel Coupled Model Transformations. Using the MultEcore tool modellers can: (i) define multi-level hierarchies using the model graphical editor; (ii) define MCMTs using the textual DSL for rule edition; and (iii) execute and analyse specific models. The execution of MultEcore models relies on a bidirectional transformation of the models into maude (Clavel et al. 2007) specifications. maude is the most efficient engine for rewriting modulo (combinations of) associativity, commutativity, and identity (Durán and Garavel 2019; Garavel et al. 2018). maude's customisation power has promoted a minimal representation distance which facilitates the bidirectional transformation between MultEcore and maude.

It has also allowed us to directly translate the multi-level setting avoiding to flatten it to two-level, which would be necessary if we wanted to use other off-the-shelf model transformation tools, such as atl—these could be adapted to MLM but not straightforwardly (Atkinson et al. 2015). Furthermore, by using maude we can directly take advantage of several of its tools for execution and analysis. When we design a multi-level DSML, we first define its syntax/structure through a multi-level modelling hierarchy and then we specify its semantics via the MCMTs. At the moment, and from an user's point of view, multi-level hierarchies are designed using a Sirius-based graphical editor for each model that conforms it. Conversely, MCMTs are specified through an Xtext-based textual DSL in a single file that contains all the rules associated with a hierarchy. Both kinds of syntax are equivalent, however, and we believe that the graphical one is more suitable for illustration and explanation of the models and rules, and therefore is the only one we use in the rest of this paper for both hierarchies and MCMT rules. But for the purpose of specifying bigger models and more complex transformation rules, we also believe that the textual syntax is more concise and manageable, and we plan to adapt it in the future for the design of multi-level hierarchies, in addition to MCMTs. A generic depiction of a node and an edge using the graphical syntax is shown in Fig. 1, for illustrative purposes. Nodes in MultEcore are depicted as yellow rectangles with the node's name on the upper part and a lower section for its attributes. The type of the node is depicted as a blue ellipse on the top border of the node, and if it has a supplementary one (see Sect. 2.2), it is depicted on the right side in a green ellipse. A red box also on the top contains the three values, separated by dashes, used by MultEcore to specify potency and therefore control the levels in which instances of the node can be created, as explained in Sect. 2.3. For edges, the name and potency are depicted together, separated by the '@' symbol, as an annotation to the arrow that represents them. The type also appears as an annotation, but in italics. Since we depict each model in our solution in a separate figure (and different editor windows in the tool), there is no need to have a graphical representation for models themselves. However, we do represent them as rectangles with their names inside in Fig. 2 to show a bird's eye view of our solution, and represent their instance-of relations with dotted arrows. Levels within a hierarchy are separated from each other by red lines, and labelled increasingly from the top, and hierarchies themselves are depicted as a dashed bounding box, in blue for the application hierarchy in our solution and in green for the supplementary one.

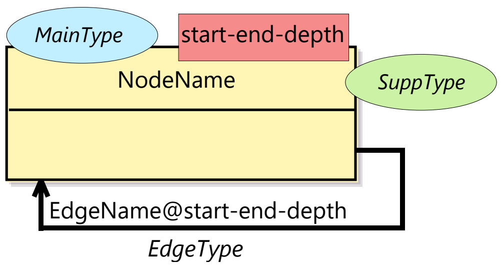

Figure 1: Node template

2.1 Levels

MultEcore is a level-adjuvant approach (Atkinson et al. 2014; Kühne 2018a) where levels are explicitly used to organise models and the elements inside them. For implementation reasons, MultEcore prescribes the use of Ecore (Steinberg et al. 2008) as root model (graph) at level 0 in all example hierarchies. However, from a theoretical point of view, any graph-based model that is able to define itself via typing would be suitable to occupy level 0, e. g., a simple graph with a node and an edge which are their own types, respectively. Therefore, we only use the types `EClass` and `EReference` for typing the nodes and edges in our models, and provide them with potency `0-*-*` (unmodifiable, and only allowed for these two elements), which enables them to type themselves, respectively, and allows their unbounded instantiation, both direct and indirect in any level below. While the implementation of MultEcore also uses EAttributes to simplify the interoperability with EMF, attributes are formally defined in our approach as nodes that are double-typed as data types. The specific mechanism is out of the scope of this paper, but the interested reader can find a detailed explanation of this mechanism in Macías (2019, Sect. 2.3.3). We omit Ecore and the level 0 from the figures in this paper for the sake of simplicity.

Models in MultEcore are distributed in *multi-level modelling hierarchies*. A multi-level modelling hierarchy in our context is a tree-shaped arrangement of models with a single root at the top of the hierarchy tree. Levels are indexed with increasing natural numbers starting from the uppermost one, having index 0. All our inter-level relationships between models, nodes and edges are represented via typing relations with the 'instance-of' meaning. We use levels as an organisational tool, where the main rationale for locating elements in a particular model is based on how they could potentially define an independent modular artefact. In this regard, we encourage the *level cohesion* principle (Kühne 2018a), that is, we recommend to organise elements that are semantically close (by means of potency and level organisation). On the contrary, we do not promote the *level segregation* principle, which establishes that level organisational semantics should be unique, i. e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. Still, we generally use typing relations with classification semantics, and the typing relation still implies that a node defines which attributes its instances can instantiate and which relations they can have to other nodes. Furthermore, the MultEcore tool checks correct potency and typing safeness. Typing safeness is checked via internal constraints that forbid relations to be circular, reversed or inconsistent neither vertically, i. e., within the same hierarchy, nor horizontally, i. e., if we consider more than one hierarchy.

2.2 Supplementary hierarchies

Frequently, we denote a multi-level hierarchy as the *main* or *default* one and call it *application hierarchy*, since it represents the main language being designed. An application hierarchy can optionally include an arbitrary number of *supplementary hierarchies* which add new aspects to the application one. There exist other techniques to achieve the composition of languages or the inclusion of additional aspects into a main language (Méndez-Acuña et al. 2016), such as *merge*, *weaving*, *inheritance* and *facet-oriented modelling* (Lara et al.

2018). Our approach based on supplementary hierarchies, present some advantages w.r.t., for example, a merge operator. In the merge approach, a new element is constructed upon the original elements that are going to be merged. This promotes the loss of the original elements that have been merged. This capability might be useful in several situations, specially when the elements that are being merged are not identical, but powering up each other. With the supplementary hierarchies technique, we can use the individual elements as well as use the composed one, increasing the number of resources available for the modeller. We say that we achieve a *virtual composition*, rather than a *physical composition* (such as the merge, weaving and inheritance approaches). Virtuality refers to the capability of dynamically adding and removing new types to elements in a non-intrusive way. The facet-oriented modelling approach (Lara et al. 2018) shares similarities with our approach. Some key features that both approaches support are: (i) modularity and non-intrusiveness; (ii) capability of easily extending elements with new types and attributes; (iii) manual and automatic acquisition of types and attributes; and (iv) mechanisms to control the repetition of features between the composed languages.

Adding or removing supplementary hierarchies is made possible by the incorporation or extraction of additional typing chains (see Wolter et al. (2019) for the formal details). For instance, we might have different hierarchies (physically separated, e. g., different projects in the MULTECORE tool) that we want to use together. Such a result can be achieved by assigning the role of application hierarchy to one of them and adding the rest as supplementary ones. In this paper, the *Process Hierarchy* acts as application hierarchy and the *Timestamp Hierarchy* is a supplementary one (see Fig. 2 and Sect. 4).

2.3 Instance Characterisation

MULTECORE allows for deep characterisation (Atkinson and Kühne 2001) which means that the elements of a model can be instantiated not only in the model immediately below it, but also further down in the hierarchy. It is common in level-adjuvant approaches to use the so-called *potency* mechanism to control the deep instantiation. Potency (Kühne 2018b) is a well-known concept in MLM and it is used on elements as a way of restricting the levels at which this element may be used to type other elements. By using potencies, we can define the degree of flexibility and restrictiveness that we want to allow on the instantiation of the elements of a multi-level hierarchy. Our potency specification is composed by three values on nodes and edges and by two values on attributes. The first two values, *start* and *end*, specify the range of levels below, relative to the current level, where the element can be directly instantiated. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, or re-instantiated, regardless of the levels where this occurs. Since attributes can be instantiated only once, as it does not make sense to create an instance of such instance, the depth on attributes is always 1 and it is not modifiable. Hence, in practical terms, only the first two values (start and end) of the potency are available to the user.

It is worth mentioning that in some parts of this paper, for abbreviation purposes, we use the `X:Y@n` notation to represent that `X` is an instance of `Y`. The optional `@n` represents the `n` number of levels in which `Y` is located above (to which we informally refer as *reverse potency*), with respect to `X`. Note that the default case is `@1`, which is omitted.

3 Analysis

In this section we discuss our interpretation of the case description, clarifying the assumptions and additions that we have considered to the original description.

First, the challenge description states that '*domain-specific concepts may be defined in their dedicated branches of a hierarchy of models without polluting the general terminology of process management, allowing domain-specific behaviour to be defined for each branch of the hierarchy* *while allowing for the reuse/enforcement of common structure/behaviour*'. This is precisely the way in which we have organised the required concepts: in a hierarchy with a top model for the generic elements related to processes (Fig. 2, top), from which two branches bifurcate. The first branch (Fig. 2, right) refines these concepts for the (sub)domain of software engineering processes. The second branch (Fig. 2, left) does the same for the domain of insurance processes, since we have also included this optional set of concepts in our submission. Based on the suggested refinements (instantiation) of concepts in both branches, the software branch has one more level, since intermediate refinements (e. g. SEActor) are required.

Second, the description also requests that '*submitted solutions should include bottom-level instances, at least for key types, exemplifying all attributes mentioned in the challenge description*'. So both of our branches include a bottom-most model which illustrates the instantiation of the nodes, edges and attributes to define a specific state of a process. The result is a five-level, two-branch hierarchy, where each level accounts for a different degree of abstraction in the challenge description. The four user-defined levels (ignoring level 0 with Ecore) are closely aligned with the ones proposed by de Lara et al. in the original process case study (Lara and Guerra 2018, Fig. 4).

Third, we assume that when we declare a concept—usually represented with a node—as a meta-concept that will be used to later define actual realisations of it, the former is clearly acting as a type. For instance, the challenge description states the need to define '*actor types*' so that we can define '*actors*'. Hence, we consider that naming the meta-concept ActorType is redundant, and therefore we choose to simply name it Actor. Consequently, all the generic elements such as actor type, task type and process type, are named Actor, Task and Process in our solution. It is worth mentioning at this point that we also use a naming convention for relations among nodes, in which verbs are always used in the third-person singular form.

Fourth, we understand that the relation between actors (people) and the duties they can fulfil (roles) is an N:M relationship. That is, one person may have more than one purpose in a process, and one purpose may be shared between several people. If we were to model this situation with actors being connected via a relation to tasks, we would be forced to explicitly model all N×M permutations of people allowed to do tasks, creating too much redundancy. In order to avoid this issue, we created a distinction between actor and role, and created the actor - role - task triangle of concepts, which addresses P5, P9 and P14 (see Sect. 5). This also allows us to easily apply a composite pattern (Gamma et al. 1994) for combined roles, which are suggested in P15. More importantly, these elements do not affect the general semantics of the models or the alignment with the requirements of the challenge.

Finally, as discussed in Sect. 5, we chose to create a secondary hierarchy to support the requirement for time stamps (P19) in a minimally invasive manner. We also argue that this scenario is a perfect fit to such kind of *aspect orientation* techniques for MLM that are supported in MultEcore.

4 Model presentation

Fig. 2 shows the overview of the system architecture we have constructed. We first detail the (main) application hierarchy that captures both domains described in the challenge. This is represented within the dashed central box in the figure, under Process Hierarchy. We describe each level in a subsection and start from level 1 (process).

We also describe how we use the supplementary dimension (dashed box to the right) Timestamp Hierarchy to address one of the requirements of the challenge. Note that supplementary hierarchy is not bound to a specific level but it is orthogonal to the application hierarchy, which means that several models within the Process Hierarchy can make use of the types and attributes defined, in this case, in the unique timestamp model. We further describe this supplementary hierarchy in Sect. 4.3.3.

We only display the cardinalities on edges in those cases where it is not the default one (0..*). The reason for this value as default is due to MultEcore's focus on flexibility, where such values are generally those which allow more kinds of instantiation. This principle also applies to the value of potency depth for nodes and edges, which is unbounded by default and therefore enables the unlimited instantiation of those nodes and edges unless otherwise specified. It is also important to note that we do not show the Ecore model located at level 0, but start from level 1 as shown in Fig. 2.

4.1 Level 1 – Process

The first model in level 1 contains the concepts concerning universal processes (see Fig. 3) and corresponds to the process item placed at the top in Fig. 2.

A Process contains an arbitrary number of Tasks. As shown in the figure, the type of a node, provided by some element in an upper metamodel, is indicated in a blue ellipse at its top left side, e. g., EClass is the type of Process. Notice the

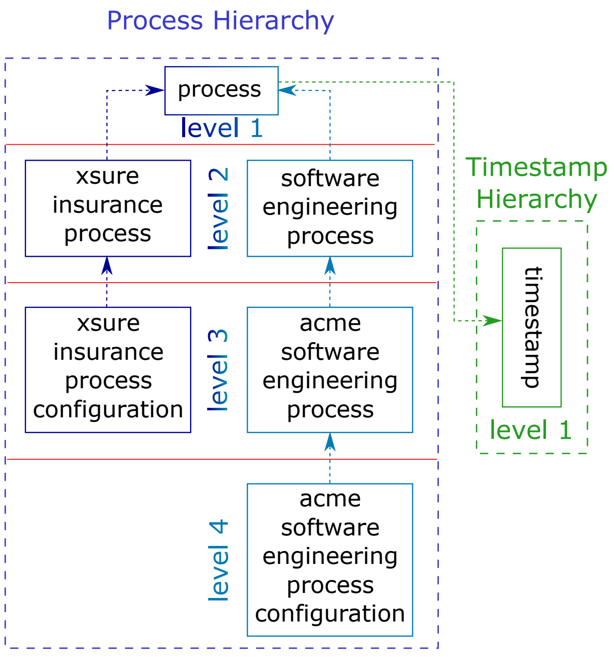
Process Hierarchy
process
level 1
xsure insurance process
level 2
software engineering process
xsure insurance process configuration
level 3
acme software engineering process
level 4
acme software engineering process configuration
Timestamp Hierarchy
timestamp
level 1

Figure 2: High-level system overview

second green ellipse at the right of Process that provides it with a supplementary TimeStamp type. The supplementary type is always placed to the right of the node, while the application one is placed at the top left corner of it so they can be distinguished. Even though we further discuss this in Sect. 5, it is worth mentioning that this second type enables us to instantiate the lastUpdated attribute on any node of the process hierarchy. Note that all the elements within the process hierarchy that are illustrated in this section instantiate the lastUpdated attribute with value 26-Apr-21. All elements in this model include this second type and can therefore instantiate the attribute, same as their instances in the levels below.

The type of an arrow is written near the arrow in italic font type, e. g., EReference for contains. We support attribute declarations that can be typed by one of the four basic Ecore data types, namely Integer, Real, Boolean and String. For instance, Task has declared four attributes, beginDate and endDate of type string, expectedDuration of type int and isCritical as a boolean. Nodes can have at the same time declared and instantiated attributes, as illustrated in Task, that has the four declared attributes commented above, and the lastUpdated instantiated attribute. The annotations displayed as three numbers in a red box at the top right of each node, and concatenated to the name after @ for every edge, specify their potencies. Potency in attributes is displayed as two numbers as an attribute's depth is always 1, since first it is declared, and it can be instantiated only once in a level below. For instance, the potency specified for Task is 1-2-3, which means that an element can be directly instantiated one and two levels below (levels 2 or 3 in the hierarchy), and such instances can be re-instantiated up to 3 additional times. This depth is therefore dependent on the value of the type, and the depth of an element must always be strictly less than the depth of its type.

Each process must have one and only one Initial Task (1..1 cardinality in initialTask edge) and can have one or more Final Tasks (1..* cardinality in

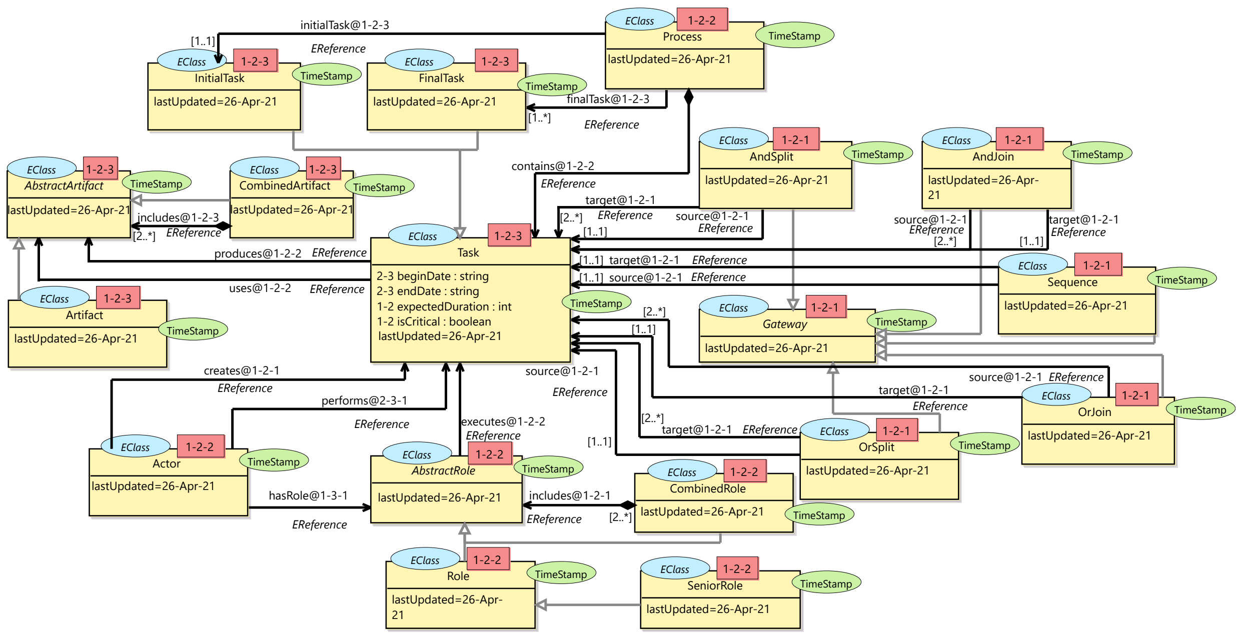
EClass 1-2-2 Process lastUpdated=26-Apr-21 TimeStamp
initialTask@1-2-3 EReference [1..1]
EClass 1-2-3 InitialTask lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-3 FinalTask lastUpdated=26-Apr-21 TimeStamp
finalTask@1-2-3 EReference [1..*]
contains@1-2-2 EReference
EClass 1-2-1 AndSplit lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-1 AndJoin lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-3 AbstractArtifact lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-3 CombinedArtifact lastUpdated=26-Apr-21 TimeStamp
includes@1-2-3 EReference [2..*]
target@1-2-1 EReference [2..*] [1..1]
source@1-2-1 EReference
EClass 1-2-3 Task 2-3 beginDate : string 2-3 endDate : string 1-2 expectedDuration : int 1-2 isCritical : boolean lastUpdated=26-Apr-21 TimeStamp
produces@1-2-2 EReference
uses@1-2-2 EReference
EClass 1-2-3 Artifact lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-1 Sequence lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-1 Gateway lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-1 OrJoin lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-1 OrSplit lastUpdated=26-Apr-21 TimeStamp
creates@1-2-1 EReference
performs@2-3-1 EReference
executes@1-2-2 EReference
EClass 1-2-2 Actor lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-2 AbstractRole lastUpdated=26-Apr-21 TimeStamp
hasRole@1-3-1 EReference
includes@1-2-1 EReference [2..*]
EClass 1-2-2 CombinedRole lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-2 Role lastUpdated=26-Apr-21 TimeStamp
EClass 1-2-2 SeniorRole lastUpdated=26-Apr-21 TimeStamp

Figure 3: Level 1 – Process model

finalTask edge). The MultEcore tool allows us to make use of the *inheritance* relation. The inheritance (i. e. specialisation) relation is a special type of arrow among any two nodes within the same level, which imposes on the child node the same typing and potency as the parent node. Moreover, the inheritance relation gives the child node access to the incoming and outgoing arrows of the parent node together with its attributes, while still allowing the child node to define additional attributes or arrows. For instance, InitialTask and FinalTask are children nodes of Task. In MultEcore we can also mark a node, e. g., Gateway, as an abstract node, which cannot be instantiated (indicated by the name in italics). This means that a Gateway must always be instantiated using one of its five children, namely, AndSplit, AndJoin, Sequence, OrJoin or OrSplit (right side of Fig. 3). They can connect one or more tasks, depending on the gateway, as indicated in the multiplicity in each source and target relations.

Tasks are created and performed by Actors. We define *actor types* as Roles (as discussed in Sect. 3). The edge hasRole between Actor and AbstractRole models this. We apply to roles the Composite pattern from object-orientation (Gamma et al. 1994). We define AbstractRole as an abstract node. Normal roles are defined as Role and further special roles might inherit from it, for instance, SeniorRole inherits from Role. Furthermore, we use CombinedRole to define roles than can be composed by simple roles (the 2..* cardinality in the includes edge ensures that there are at least two roles combined). Also, certain roles can perform certain tasks, which is covered by the executes edge from AbstractRole to Task. Regarding artefacts, the composite pattern is applied again so that an AbstractArtifact can either be a simple Artifact or a composition of more than one of them into a CombinedArtifact, as indicated by the includes relation. A Task might use and/or produce artefacts of any of these two kinds.

4.2 Software engineering process domain

In this section, we disclose the domain-specific aspects for the software engineering process which corresponds to the right hand branch of the application hierarchy (see Fig. 2).

4.2.1 Level 2 – Software engineering process

This level concerns the refinement of concepts from general processes that apply to any software engineering domain. It is represented in Fig. 4 and it corresponds to software engineering process in Fig. 2.

The creation of this level facilitates the resolution of multiple requirements which are discussed in Sect. 5. Every software engineering artifact SEArtifact, which is typed by Artifact (placed at level 1, Fig. 3) must have a responsible software engineering actor (responsibleActor relation with multiplicity 1..1 to SEActor). The specification of SEArtifact and SEActor forces the definition of any artifact or actor within the software engineering domain to be typed by SEArtifact or SEActor instead of the generic Artifact or Actor, respectively. Also, each concrete SEArtifact must be assigned a version number (versionNumber attribute). Note the potency 2-2, as the instance level of the software engineering domain is placed at level 4 (acme software engineering process configuration at the bottom of Fig. 2). Furthermore, SEValidationTasks have to validate at least one SEArtifact, represented via the validates reference with cardinality 1..*.

4.2.2 Level 3 – Acme software engineering process

We now discuss the aspects related to the Acme software engineering process. This model corresponds to the acme software engineering process component in Fig. 2. We show in Fig. 5 selected parts of the model (right-hand side) in order to compare it with the graphical representation in concrete syntax given in the Challenge description (left-hand side of Fig. 5). The complete model that fulfils all the requirements and specifications is shown in Fig. 6. From this point on, the elements we describe refer to the right-hand side of Fig. 5.

At this level, we specify the *types* that belong to the Acme software engineering domain. Note that some of the types of the elements, e. g., Ini-

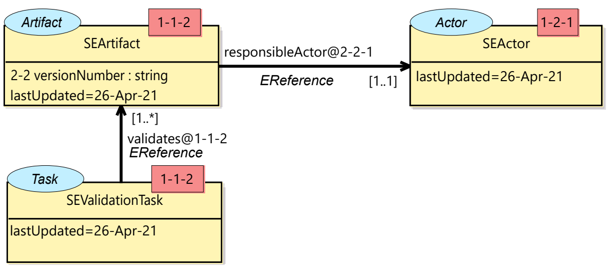

Figure 4: Level 2 – Software engineering process model

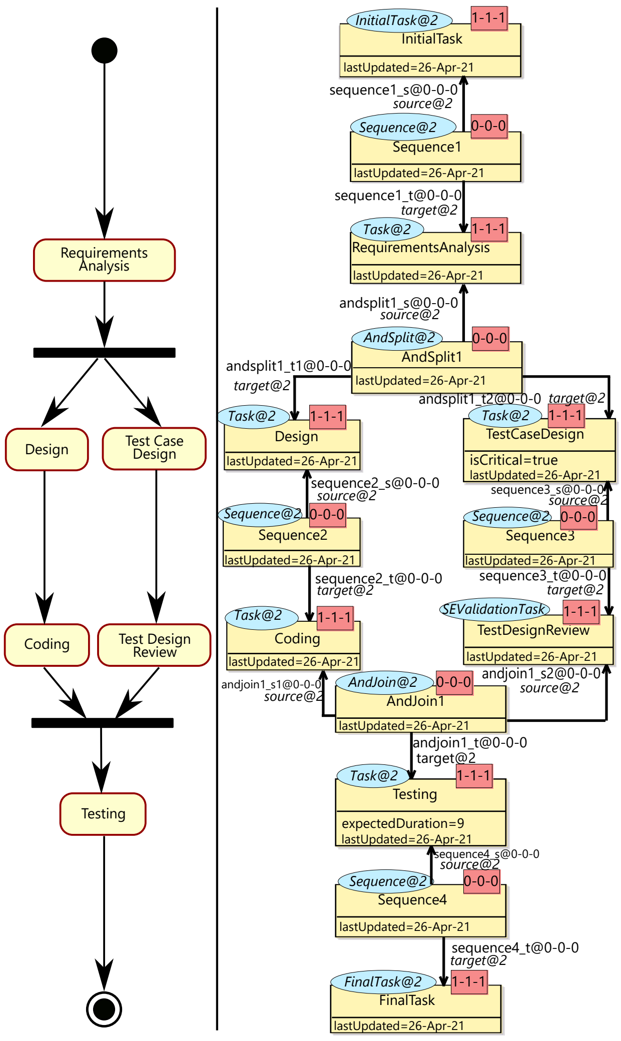

Figure 5: Level 3 – Selected parts of the Acme software engineering process model (right-hand side) to compare it with the graphical schema given in the Challenge description (Almeida et al. 2021)

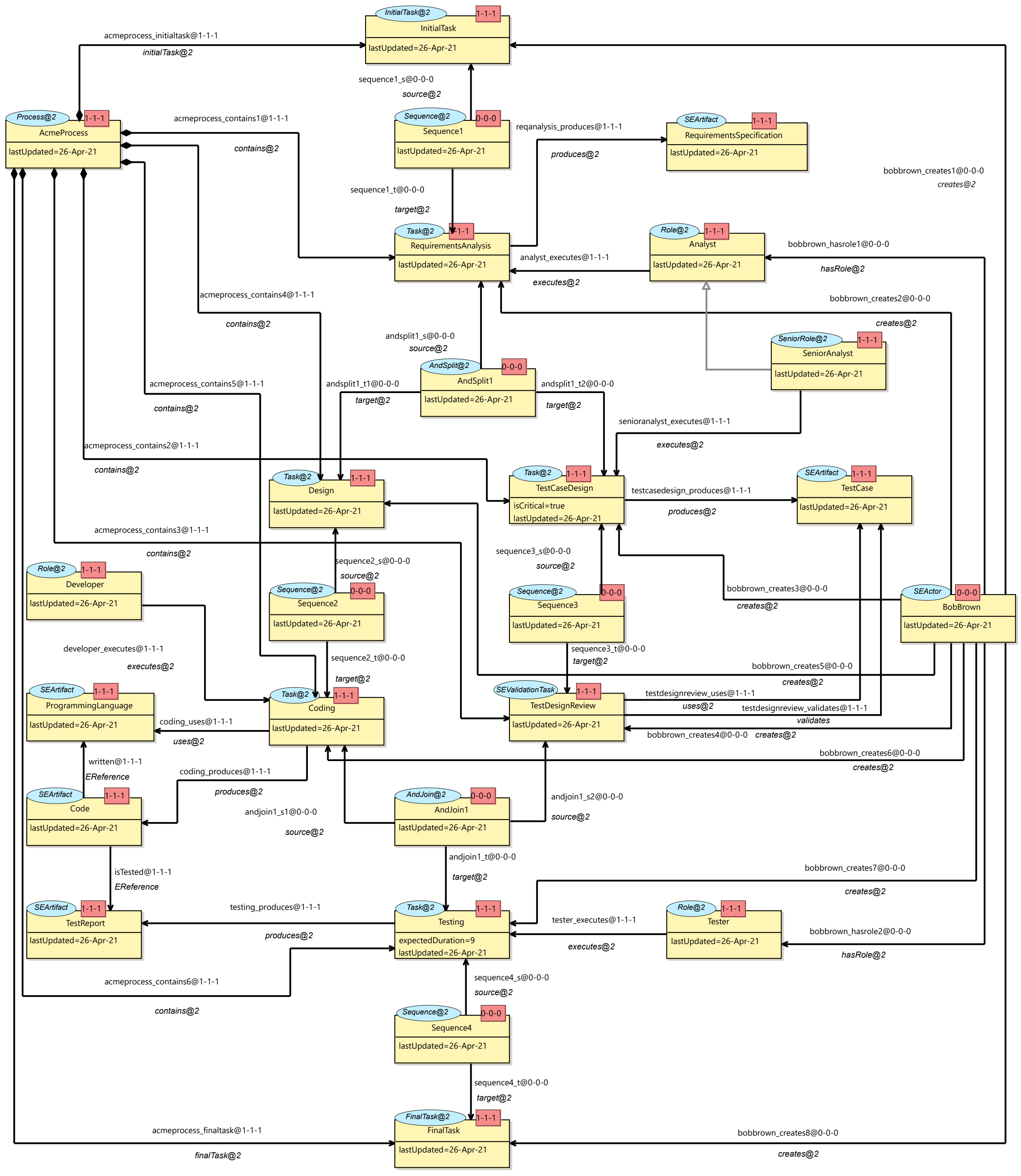

Figure 6: Level 3 – Complete Acme software engineering process model

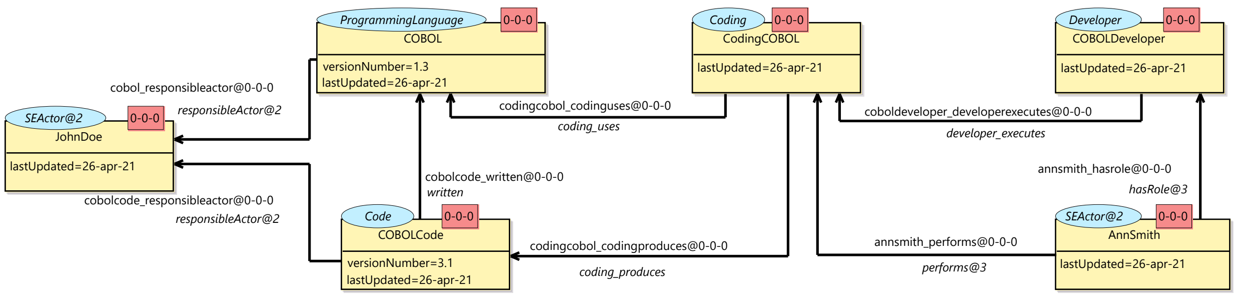

Figure 7: Level 4 – Acme software engineering process configuration model

tialTask, Sequence1, RequirementsAnalysis, etc., are allocated two levels above, which is specified for nodes in the ellipses where the type is given concatenated with @2, and for the edges concatenated in the types with italic font. One can observe, comparing it with the left-hand side representation, that all the information is accurately reproduced and easy to track.

Note that the potency of some elements, such as Sequence1, AndSplit1, Sequence2, Sequence3, AndJoin1 and Sequence4, is 0-0-0. These values are due to the fact that those elements cannot be further instantiated, which clearly indicates that they belong to this level where the general Acme workflow is represented. Also, notice that we instantiate some attributes here apart from lastUpdated, for example isCritical, which is set to true in TestCaseDesign node, and expectedDuration=9 in Testing node.

It could be argued that common concepts of software engineering processes like Coding and Testing should belong to the model above this one (level 2, software engineering process) to simplify the definition of other software processes within Acme or other companies. In such a way, each specific process could refine these concepts as needed or use them as a standard definition of common activities, and the software branch of our solution could have sub-branches for different software processes and their configurations. While this is a valid option that we considered for our submission, we chose not to implement it since it is not required for the scenario presented in the requirements. We believe that abstracting those concepts in this case would mean over-engineering the solution and would harm the simplicity and readability of our models. A different example of this kind of abstraction of commonalities is shown for more abstract concepts of processes in the process model (Fig. 3). Of course, if the requirements changed to allow for the definition of different software processes, our solution could be modified accordingly by ‘lifting’ the aforementioned concepts to level 2.

4.2.3 Level 4 – Acme software engineering process configuration

In this section, we describe the aspects related to a specific application of the concepts defined on the Acme software development process. In this branch (software engineering domain) this model represents a state (i. e. a potential execution) of a process. This model is depicted in Fig. 7 and corresponds to the acme software engineering process configuration element in Fig. 2. The nodes and relations displayed in this model have been reconstructed using information provided along the software engineering domain requirements (S1, S2, etc.) from the challenge description (Almeida et al. 2021). Notice that all nodes and relations at this level have as potency 0-0-0, since this is the bottom-most model and cannot be further instantiated. We discuss the elements of this model from left to right on Fig. 7.

JohnDoe (typed by SEActor@2) is responsible of the concrete artifacts COBOL (typed by ProgrammingLanguage, with versionNumber=1.3) and COBOLCode.

This responsibility is indicated by the incoming cobol_responsibleactor and cobolcode_responsibleactor edges. Also, COBOLCode is written (the type of the edge cobolcode_written) in COBOL. Besides, CodinCOBOL:Coding task uses COBOL and produces COBOLCode. These two relations are captured by codingcobol_codinguses and codingcobol_codingproduces relations, respectively.

Finally, AnnSmith:SEActor@2 is an actor that has assigned, via the annsmith_hasrole:hasRole@3 relation, the COBOLDeveloper role. AnnSmith performs CodingCOBOL, which the COBOLDeveloper role is allowed to execute. Notice that certain types of the elements aforementioned (e. g., Developer or ProgrammingLanguage) are not explicitly shown in the excerpt on Fig. 5, whose full version is detailed in Fig. 6.

Another example of a model in this level can be found in Sect. 4.5, with an alternative, more complete instantiation of the Acme process.

4.3 Insurance process domain

The challenge description outlines the so-called *insurance domain*. Even though respondents are encouraged to focus on the software engineering domain, with the insurance part used '*for illustrative purposes only*', we have constructed it in a separate branch and used all the information obtained from analysing the *PX* rules and specifications given in Sect. 2.2 of the challenge description document. As one can observe in the left-hand side of the Process Hierarchy in Fig. 2, the branch is composed by two models (if we ignore process at level 1) rather than three as in the software engineering domain. The demands of this domain do not require the creation of a model that is equivalent to software engineering process model (level 2 in Fig. 2). Instead, the level 2 of the insurance branch directly corresponds to the XSure company (xsure insurance process model). This difference demonstrates the flexibility of MultEcore, where the lengths of the different branches of a hierarchy are not required to be equal.

4.3.1 Level 2 - XSure insurance process

The xsure insurance process model, which corresponds to the xsure insurance process element in Fig. 2 at level 2, represents the workflow of the ClaimHandling process. The complete model is shown in Fig. 8.

We describe the model from top to bottom. A ClaimHandling:Process is composed by all the tasks depicted in the model (nodes typed by InitialTask, Task and FinalTask) which are connected to it by containment relationships. A ReceiveClaim precedes an AssessClaim task, which are connected via Sequence2. To proceed, an AssesClaim uses a Claim:Artifact (to the middle left of the figure) and produces a ClaimPaymentDecision. Also, AssessClaim is created (via benboss_creates relation) by BenBoss, who is an Actor. Furthermore, ClaimAssessor is a Role which is allowed to execute AssessClaim tasks. We also find at the top right of the figure two roles declared, SeniorManager:SeniorRole and ProjectLeader:Role.

AssessClaim leads to the following task, AuthorizePayment, connected by the Sequence3 gateway. Both ClaimHandlingManager and FinancialOfficer roles are allowed to execute AuthorizePayment tasks. Finally, we also find, connected by sequence gateways (Sequence4 and Sequence5), the PayPremium task which comes after AuthorizePayment and that leads to FinalTask, which ends the workflow of a claim handling process.

4.3.2 Level 3 – XSure insurance process configuration

As stated before, the xsure insurance process configuration model placed at level 3 (see Fig. 2) represents the instance level in this particular branch, i. e., represents a concrete scenario and therefore a non-instantiable model (notice the 0-0-0 potencies). The model is depicted in Fig. 9.

An instance of the process ClaimHandling, named HandlingClaim123, is defined at this level, and could be interpreted as the concrete implementation of the claim assessment process of the XSure company for a claim with id *123*. It

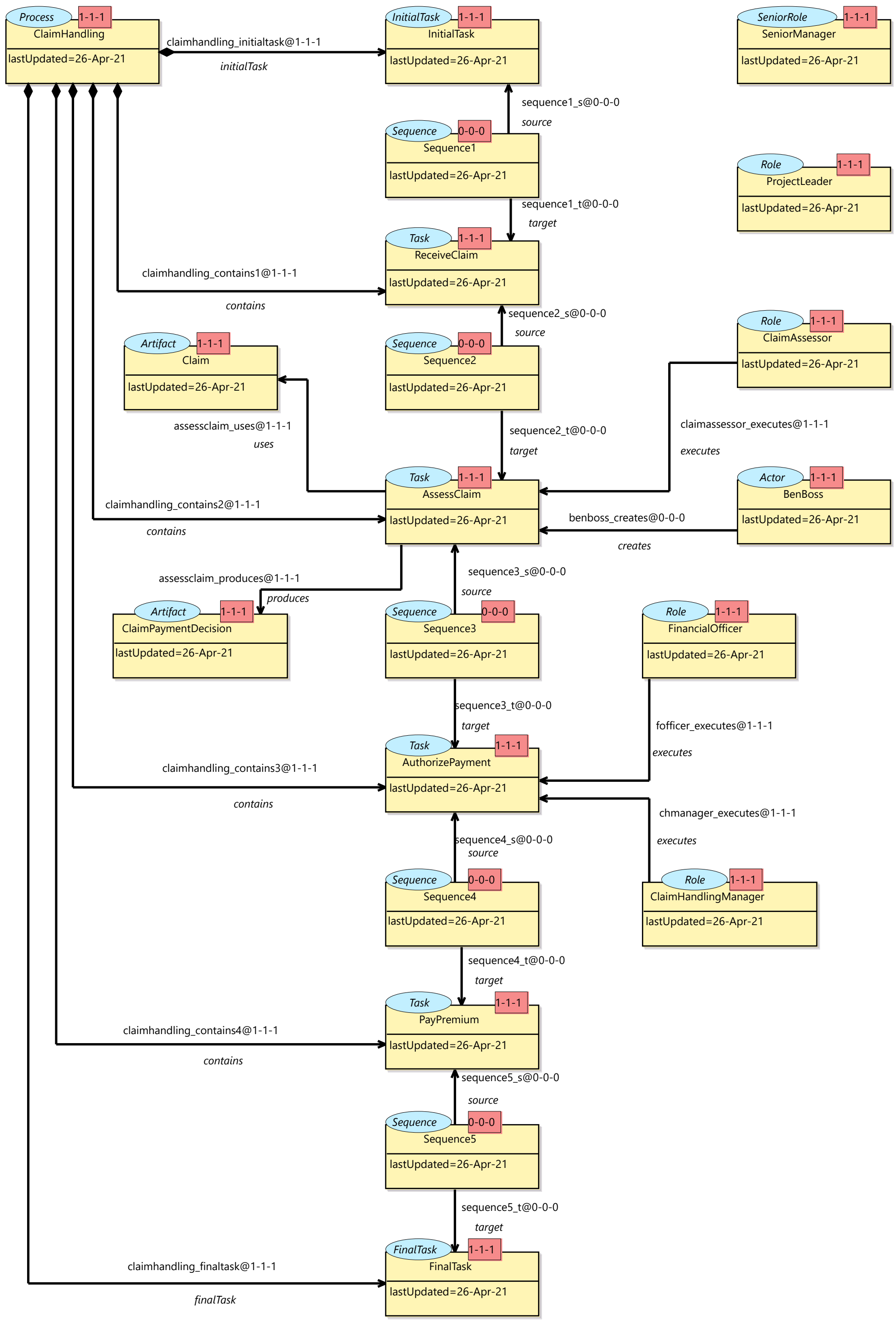

Figure 8: Level 2 – XSure insurance Claim Handling process model

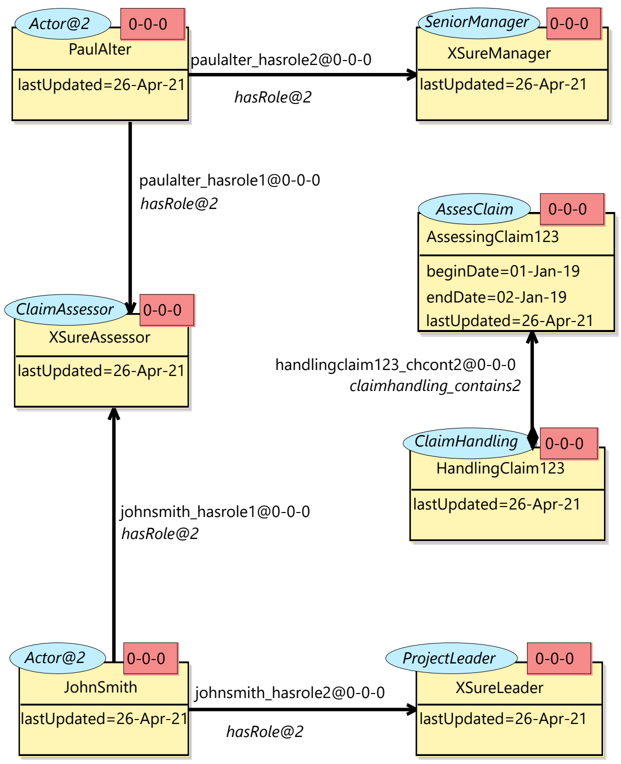

Figure 9: Level 3 – XSure insurance process configuration model

contains, via the handlingclaim123_chcont2 relation, the task AssessingClaim123 that instantiate the two attributes beginDate=01-Jan-19 and endDate=02-Jan-19. Also, XSureAssessor is a ClaimAssessor role that both PaulAlter and JohnSmith actors have assigned. Even though they share that specific role, they each have a second role, respectively, XSureManager for PaulAlter and XSureLeader for JohnSmith. This way we display that an actor might have more than one role assigned (as stated by one of the requirements).

4.3.3 Supplementary hierarchies

In this subsection, we discuss how we make use of one of the key features that characterises MultEcore. The process hierarchy, which includes both the insurance and the software engineering branches in Fig. 2, is the application hierarchy in this case. As mentioned earlier, an application hierarchy can optionally include an arbitrary number of supplementary hierarchies which add new aspects to the application one. The supplementary hierarchy notion has been applied in previous work in different ways: (i) for the runtime verification of properties of an executable workflow (Macías et al. 2018); (ii) to complement a main language with additional non-functional features, for instance, data types (Rodríguez et al. 2018) or additional information to augment the data of a node (Rodríguez and Macías 2019); and (iii) to power up instance elements, where composition of application and supplementary hierarchies could be carried out (Rodríguez et al. 2019b).

In this work, we use a supplementary hierarchy to satisfy one of the requirements of the challenge, where all elements in our solutions must have a value for the last time they have been updated. This is a perfect match for supplementary hierarchies, where we need to introduce a new aspect that affects our whole domain (process models) but does not really belong to it, since those models would still make perfect sense without such a feature, and the idea of time stamps for elements can be applied to other domains. Therefore, creating a different hierarchy which can be attached to any other hierarchy where this aspect is suitable is a more desirable alternative to polluting the existing domain (application hierarchy) with it by adding the concept to the existing models or introducing a new model which would neither make sense from an ontological point of view. Additionally, defining time stamps as a separate hierarchy eases its reusability from a tooling perspective, since MultEcore can add an existing hierarchy as supplementary in just a few clicks.

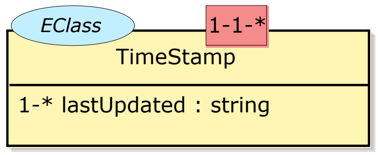

Figure 10: TimeStamp node

As illustrated in Fig. 2, we have created a supplementary hierarchy that can provide a *last updated* value to ideally any node defined in the application hierarchy. The supplementary hierarchy, called Timestamp Hierarchy in Fig. 2 consists of a single model, called timestamp which has one single node called TimeStamp, as shown in Fig. 10. This node has declared the lastUpdated attribute of type string. It is worth reminding that elements from supplementary hierarchies can be used in an orthogonal manner. The advantages of our solution by defining this feature as supplementary is that any node, in any of the models distributed along the Application Hierarchy in both branches can instantiate the attribute lastUpdated to give it a concrete value (we use the same for all the nodes, lastUpdated=26-Apr-21). The only requirement is to double-type all elements in the process model at level 1, such as Task and Gateway. The result is that any node residing in one of the models depicted in Figs. 3, 4, 5 (and its full version in 6), 7, 8 and 9, can instantiate the attribute. This attribute can be instantiated in both all nodes in level 1 and all their instances in the levels below thanks to the way we use potency. By giving TimeStamp unbounded depth, we ensure that all instances of the nodes in level 1 are also instances of it, which implies that the lastUpdated attribute is visible to them. Those instances (e. g. Code) can then instantiate the attribute even if their direct type (SEArtifact) has already done so, since lastUpdated has an unbounded end potency, which allows its direct instantiation any number of levels below. This construction does not entail that the attribute is being re-instantiated as we do for nodes and edges, which would go against the constraints of the framework; it means that lastUpdated can be *directly* instantiated in an *indirect* instance of TimeStamp.

4.4 Cross-level constraints

As introduced in Sect. 2, MCMTs can be used to specify the dynamic semantics for the definition of behavioural descriptions (as we will see in Sect. 4.5). In previous work, we have shown that this sort of semantics can be executed by using MAUDE to evolve models with the infrastructure we have built in Rodríguez et al. (2019a). However, the specification and verification of static semantics, i. e., constraints to check some structural correctness of the constructed multi-level hierarchy is explored in this section.

The usual application of the MCMTs when describing behaviour is as endogenous in-place model transformation rules (Mens and Gorp 2006). In this context, the transformation rules represent actions that may happen in the system. These model transformations (MTs) are rule-based modifications of a source model (specified in the left-hand side of the rule) resulting in a new state of such a model (determined by the right-hand side). The left-hand side (LHS) takes as input (a part of) a model and it can be understood as the pattern we want to find in our original model. The right-hand side (RHS) describes the desired modifications that we want to perform in our model and thereby the next state of the system. There is a match when what we specify in the LHS is found in our source model and the execution of the rule represents a single transition in the state space.

These transformations work fine when we want to find a match, and then produce a new state of the model. Still, this mechanism does not completely align with the one we require to define constraints. In order to be able to verify that certain constraints are satisfied we propose a *check mode* that behaves differently than conventional MTs. In this mode, the goal is to find a correspondence in the models through a two-step procedure. Instead of having a model that evolves or changes to a new state as the behaviour is specified (LHS $\rightarrow$ RHS), now, for the model to pass or to be correct with respect to the constraint, both situations (what is being specified in the LHS and the RHS) must be found in the multi-level hierarchy. The fact that the two conditions do not match (or only one of them) results in a constraint violation.

Let us analyse, for instance, the requirement *P17*: '*An actor who performs a task must be authorized for that task. Typically, a class of actors is automatically authorized for certain classes of tasks.*' Fig. 11 shows an MCMT rule in check mode to satisfy such a constraint, with a graphical notation instead of the textual one used in the tool to simplify the explanation. The META block allows us to locate types in any level of the hierarchy, and can be used in the FROM and TO blocks (separated by a black horizontal line). It is worth pointing out that the two levels specified (the one for the META and the one for the FROM and TO) in this rule are not required to be consecutive and they would match on levels 1 and 4 of the right-hand branch on Fig. 2, respectively. In the case of the insurance domain, this rule would match with levels 1 and 3, respectively, being the rule reusable for both domains.

At the META level, we mirror part of the process metamodel, defining elements like Actor, Task, AbstractRole and Role nodes and performs, hasRole, executes and executedBy edges that are used directly as types in the levels below in the rule. These are constants, which is indicated by underlining the name of the element. A constant in an MCMT rule can only match to an element with the exact name in the corresponding model that has been matched.

For variables (i. e. non-constants), we allow the type on the elements to be indirect, meaning that there can be intermediate types in the actual

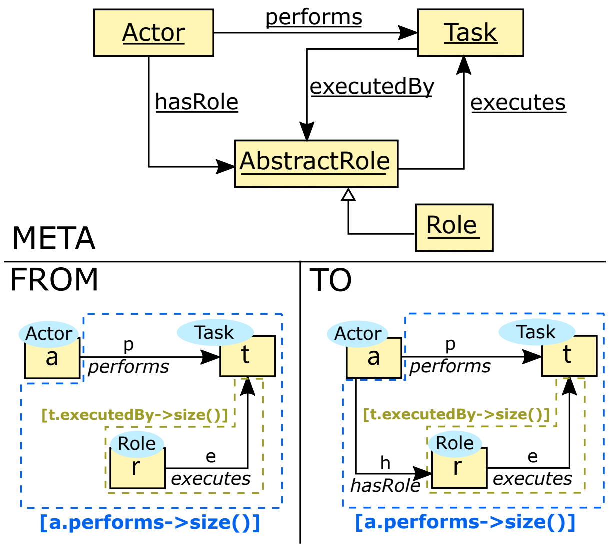

Figure 11: Constraint satisfying requirement P17

hierarchy where the MCMT is matched. We see variables in the FROM block, where a first correct match of the rule comes when an element, coupled together with its type, fits an instance of a:Actor that has a relation p:performs to an instance of t:Task. Also there must exist an instance of r:Role that is linked to t:Task via the e:executes relation. Note that there are two dashed boxes surrounding certain elements. One must take into account that in different scenarios there could an arbitrary number of tasks connected to an actor that can perform them, and that several roles can also be allowed to execute a certain task. To cover all the permutations with a single rule, we use a box-based mechanism that allows us to automatically replicate the contained elements at runtime. Boxes may appear in both sides of the rules, they can be nested, and each of them may have an explicit cardinality specified. Basic support for the Object Constraint Language (OCL) (Clark and Warmer 2003) is incorporated into the MCMTs for: (i) the computation of the cardinality of a box, i. e., the number of times it has to be replicated; (ii) for the manipulation of attribute values (not used in this rule); and (iii) for the specification of conditions (not used in this rule), which greatly improves the expressiveness of the tool. Note that while the support for OCL is very basic, we plan to extend it in future work. For instance, the expression used in the outer box ([a.performs->size()]) is using the size operation. In OCL, the size() operator calculates the size of the collection it is applied on. The a.performs expression returns the collection of edges whose source is a and its type is performs. Note, however, that the way in which types are used in MLM is a bit different than for standard OCL. This grants transitive typing, which allows for the matching candidate to have any type that is either performs itself or (in intermediate levels) other elements which are ultimately typed by performs. In practical terms, the [a.performs->size()] expression means that the size of the collection is given by the number of tasks connected to the matched actor (a) via edges of type performs. Similarly, the inner box that encapsulates r:Role and e:executes with the [t.executedBy->size()] expression would count the number of edges (which types ultimately match executedBy) the element t has. Note that executedBy relation is not defined in the process model (Fig. 3) since this rule is yet theoretical.

Once all the boxes have been unfolded for the FROM part of the rule, and there has been a match of all the (unfolded) variables, this partial matching is saved and reused in the TO block, to check whether its contents can also be matched. In the case of the TO block, for each task that an actor is performing, and given that a certain role can execute such a task, there must exist a relation h of type hasRole between the actor and some of the roles that are allowed to execute the task. The two consecutive and successful matches would verify that the multi-level hierarchy satisfies the constraint.

Note that this novel application of the MCMTs to specify constraints is highly influenced by graph constraints (Orejas et al. 2010). In fact, nested graph constraints have been widely discussed in the literature (see e. g., Arendt et al. (2014), Ehrig et al. (2006), Habel and Pennemann (2009) and Radke et al. (2018)). The current state of the MCMTs to check constraints only covers constraints in the form of IF:THEN and FOR-ALL:EXIST as explained in the previous paragraph where we refer to the two-step procedure. This is, IF we find the LHS, THEN the RHS must also be found; and FOR-ALL elements specified in the LHS (if we are using boxes) there must also EXIST what is specified in the RHS. Additionally, MCMTs can include application conditions. These are not shown in this paper but we refer the reader to our Petri nets case study[3] for an example of them.

Two more examples of constraints specified with MCMTs in check-mode can be found in the discussion of requirement P9 in Sect. 5.

[3] Petri nets case documented in the MultEcore website https://ict.hvl.no/a-petri-net-multilevel-hierarchy

4.5 Operational semantics

The challenge description does not comment on the possibility of describing the operational semantics of processes, which can also be done quite naturally through model transformations. We find this fact surprising, given that MT is one of the pillars of Model-Driven Software Engineering in general, and has been already tackled by several authors within the MLM community apart from ourselves (Atkinson et al. 2012, 2009; Kühne and Schreiber 2007; Lara and Guerra 2010; Lara et al. 2015; Rossini et al. 2014). We believe that exploring the possibilities of MT for this challenge could enrich the submissions, debate within the community and further editions of the multi-level challenge. Moreover, the process domain of this challenge is a good candidate for specifying operational semantics through MT rules, since the concept of processes being executed already implies some sort of evolution through time of the models that represent them. So, in this subsection we focus on a simplified proposal for MT rules to model the way in which gateways are triggered to create the next tasks of a process once the previous ones are completed.

For a more realistic and comprehensive collection of rules that could fully *animate* the models, a new version of the challenge would be required where MTs are taken into account to create a more complete and unambiguous description of the operational semantics of the elements on the domain.

We have explained how MCMTs could be used to specify cross-level constraints that check the structural correctness of the multi-level hierarchy (Sect. 4.4). Now we describe how MCMTs can be used to specify the behavioural descriptions of the modelled system by means of model transformations. MCMTs have been widely improved since their initial proposal in Macías et al. (2019). While MCMTs are powerful enough to describe many behavioural aspects, it is necessary to have an engine that can execute them against a multi-level hierarchy to have an actual execution mechanism capable of evolving the models. To do so, we rely on the MAUDE System (Clavel et al. 2007; Durán et al. 2020), a specification language based on rewriting logic (Meseguer 1992), which can naturally deal with states and non-deterministic concurrent computations. A preliminary version of the infrastructure we have implemented (still under development) was presented in Rodríguez et al. (2019a). In that version, the multi-level hierarchy and the set of MCMT rules was transformed into a functional MAUDE representation that could be executed using the MAUDE console environment. The obtained results in MAUDE, i. e., the new model states produced, where stored in separate XML files. These had to be manually taken one by one and interpreted by MULTECORE, which had some practical and usable limitations. Also, the MCMTs expressive power was rather limited in that version compared to the capabilities they offer nowadays.

Since the goal of this paper is to demonstrate how MULTECORE can be used to model the proposed challenge, we do not enter into the MAUDE specification details. Still, all the produced MAUDE files can be found in our GitHub repository. The current state of the infrastructure that connects MULTECORE with MAUDE relies on a bidirectional transformation that takes the entire MULTECORE representation (both the multilevel hierarchy and the associated MCMT rules) and automatically generates MAUDE specifications. Then, this transformer takes the XML output files that MAUDE produces as result of performing execution, and automatically translates them back into MULTECORE models that are graphically displayed. The MAUDE part is handled by a background process which makes the underlying MAUDE transformations transparent to the user. In other word, the integration of MAUDE as a process within MULTECORE allows us to execute and verify the instance models directly from the MULTECORE interface and obtain new results automatically.

To show the potential of the MCMTs we provide now an example of how they can be used to systematically create parts of the models based on the information allocated within the process hierarchy. Therefore, as an illustrative example and to open this line for future Multi-Level Modelling

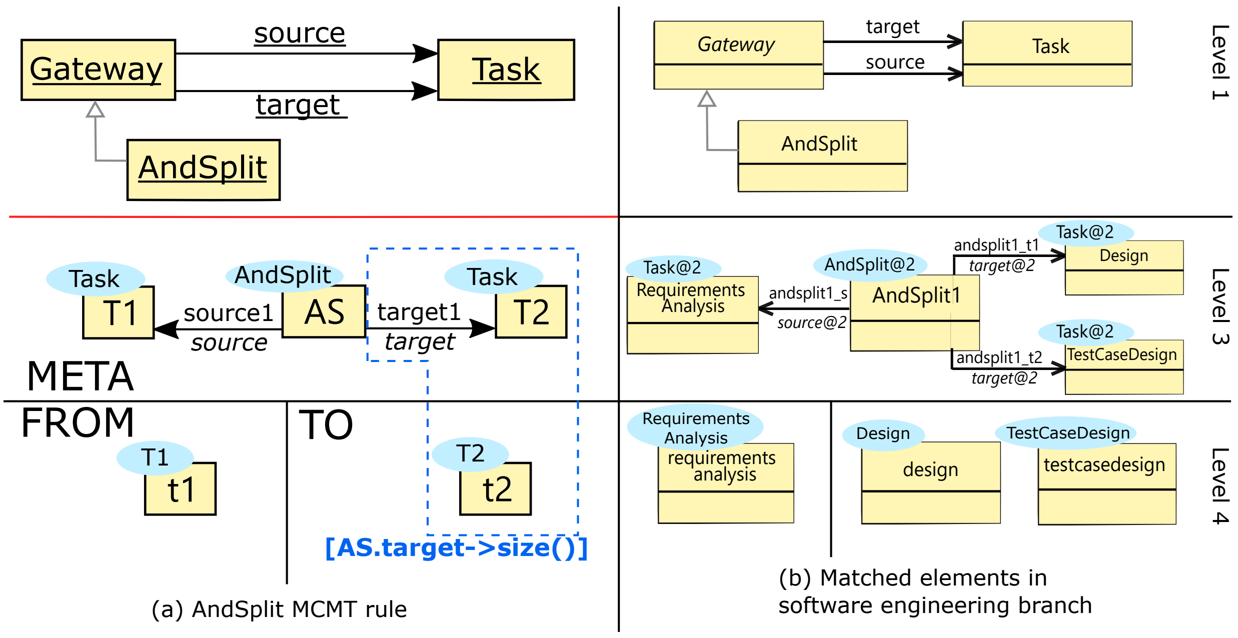

Figure 12: (a) MCMT rule to create new tasks based in their types connected via and-split gateways. (b) Matched elements in the Software engineering branch

challenges, we have sketched five simple MCMT rules that involve the creation of new tasks at the bottom-most levels (level 3 for insurance and level 4 for software engineering) through the information of corresponding gateways connected to the tasks on the levels above. This set of rules handles several cases regarding the different gateways and the initial and final tasks. In the following, we describe one of the rules, but the remaining ones can be found in their textual form with the rest of the artefacts in our solution to the challenge on GitHub.

We show in Fig. 12(a) an MCMT rule to create several output tasks from an input task where their types are connected via an *and-split* gateway. Similarly to the rule shown in Fig. 11 in Sect. 4.4, we define at the topmost level of the MCMT constant elements such as AndSplit that inherits from Gateway that is connected to Task via source and target relations. The second META level defines variable elements, such as T1 and T2 of type Task and AS of type AndSplit that connects with the former two via source1 and target1, respectively. These will be matched with elements in level 3 of the software engineering branch, and with elements in level 2 for the insurance branch. In the FROM block, we identify a single t1 task whose type (T1) would be the input of the corresponding and-split gateway. Then, in the TO block, we remove the matched t1 and create the new t2 tasks, the types of which are the outputs of the and-split. Note that each particular process can establish an arbitrary number of output tasks for different instantiations of AndSplit. To make a generic rule that works for any number of output tasks, we define boxes around target1 and T2 from the META block and around t2 from the TO block. Note that the possibility to establish cross-level boxes is a new feature which we have introduced in MultEcore to be able to handle the current case. The OCL expression [AS.target->size()] counts how many target tasks are connected to the matched and-split gateway. Note that the cross-level box is needed because the gateway information is not given at the instance level, but a level above. Therefore, the three elements must come together into the same box, so when it is unfolded at runtime, the type of each produced t2 is paired correctly with the information located in the level above.

Fig. 12(b) shows the corresponding matched elements in the software engineering branch. We can observe in this example the vertical flexibility of the MCMT rules, since the matched elements are distributed within level 1, level 3 and level 4 for the three levels specified in the rule. Even though there exists an intermediate level in the software engineering branch (level 2), it is not a problem for the rule to ignore it and match the appropriate elements in the correct models. At the bottom of Fig. 12(b) we see, divided by a vertical line, the two parts of the model that would match the FROM and the TO blocks. In this case, requirementsanalysis would match t1 and design and testcasedesign the two replicated t2 variables.

The MAUDE integration within MultEcore allows us to use all the available tools for MAUDE. For execution, we allow the modeller to specify a number of steps to be executed (being each step the application of one of the available rules) or directly customise the execution by stating which rules and in which order should they be applied. To demonstrate the application of different rules, we have created two basic instance models, one for the insurance domain and one for the software engineering domain, each of them with a single element named initialTask. The five MCMT rules specified allow us to reach a finalTask based on each of the workflows defined in Figs. 6 and 8. Note that the executions are only concerned about tasks and gateways, and do not consider other elements such as artefacts or actors.

Fig. 13 shows the models for each of the seven execution steps that have been obtained by applying each corresponding MCMT rule on the software engineering domain. At the top left, we have the initial model acme-execution-step-1 with the initialTask node. The reasons for having the first model as the one containing the initial task are two: (i) having a rule to create the initial task from an empty model would require a complex left-hand side and negative application conditions to ensure that is not always triggered, since we would need to ensure that the model is empty first; and (ii) if we visualise the evolution of the process execution as a sequence of models, having an empty model as the first step does not provide any meaningful information, hence we choose to avoid it. To its right, obtained by applying the rule that triggers the `Sequence1` gateway we have the `acme-execution-step2` with the `requirmenetsanalysis101` node. Note that the number appended to the name is generated using a *Counter* object that is used in the MAUDE representation, whose value gets increased every time a new identifier is created. This counter allows us to create fresh elements avoiding name duplication. The name below each curved arrow between model states (instances) does not represent the name of the executed MCMT rule, but the gateway that is matched in the level above where the workflow is represented (see Fig. 6). One can observe at the bottom right of Fig. 13 how we finally reach an instance `finaltask101` representing the end of the workflow.

Likewise, and demonstrating the horizontal flexibility of the MCMTs, Fig. 14 represents six model

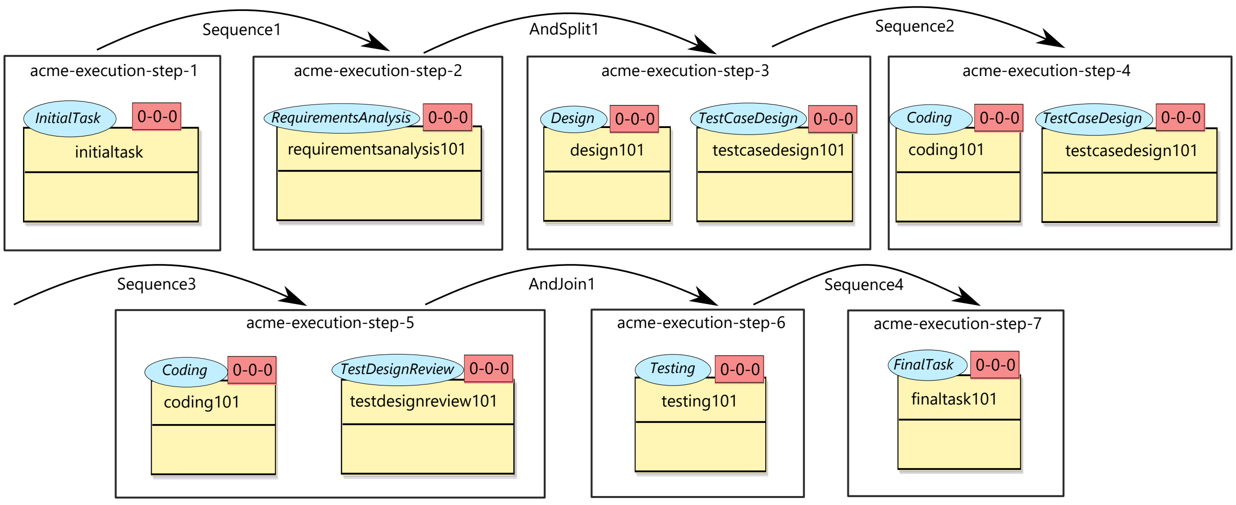

Figure 13: Acme software engineering obtained states by applying subsequent MCMT rules

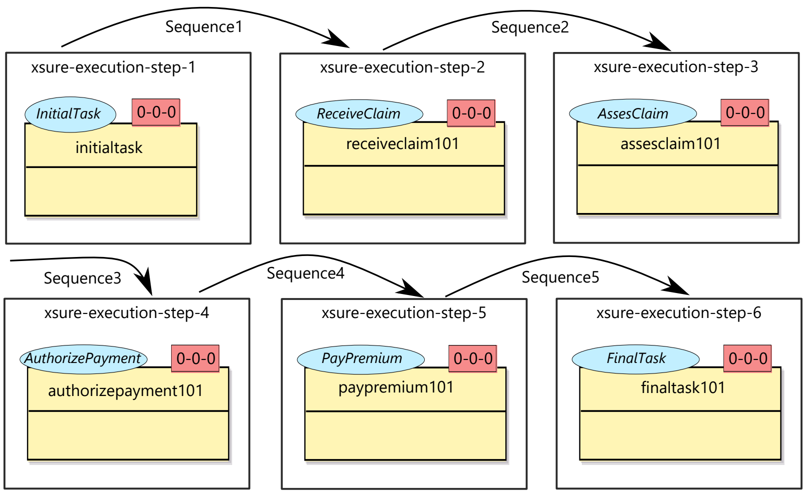

Figure 14: Xsure insurance claim handling process model states obtained by applying subsequent MCMT rules

states produced by the execution engine by applying the same set of rules to the insurance domain. Similarly, the workflow established in the corresponding level above (the XSure insurance claim handling process) is defined in Fig. 8. The process is quite similar as for software engineering, where there exists an initial task (top left of Fig. 14) in the initial model `xsure-execution-step-1`. Then, by applying rule by rule we get new elements in new model instances, such as `receivclaim101`, `assesclaim101`, . . . and finally `finaltask101`.

An alternate set of rules could be defined for a slightly different operational semantics: we could exclude the bottom levels in both our branches, or have the bottom level be a replica of the level above (including all gates), where we indicate the current state of the process execution via attributes—e. g., a *current* flag for active tasks—, pointers—e. g., a *Current* node and edges that connect it to the active tasks—or tokens in a similar fashion to Petri nets. These are all valid alternatives that we discussed for our solution and have used in previous works (Durán and Rodríguez 2021), but we believe that the one presented in this subsection is more suitable in the context of the process challenge, for the following reasons:

- Our approach does not require that the bottom level is a duplicate of the level above, thus avoiding duplication of elements.
- We do not need to introduce new elements in the models which would be not ontological but instrumental for the execution, and therefore would pollute the model.
- The current state of an execution is visible at a glance in latest model generated by MCMT rule application.
- The full execution history is preserved as a sequence of models containing the finished tasks, allowing traceability, backtracking, etc.
- The MCMT rules required by this approach are simpler and easier to understand, especially in textual syntax.

It is important to point out that our modelling hierarchy is not influenced by the way in which its semantics are specified. That is, we could easily specify different MCMTs to change the way in which a process model is executed. In such hypothetical scenario, an instance could grow as the process advances, with a single model capturing the full history of the execution. That model is shown in Fig. 15, and it provides a complete view into a finished execution of the Acme process from Fig. 6.

5 Satisfaction of Requirements

In this section, we explain how our solution addresses all the requirements in the challenge description. First, we discuss the ones related to the more abstract concepts of processes, tasks, actors and artefacts (Almeida et al. 2021, Sect. 2.2.). We preserve their original name format (PX, with X being a number) for easy traceability and reproduce their text for self-containment.

P1

‘*A process type (such as* claim handling*) is defined by the composition of one or more task types (receive claim, assess claim, pay premium) and their relations.*’

This requirement is addressed with the definition of the nodes `Process` and `Task`, and the containment relationship from the former to the latter. They are contained in model `process` at the top of the hierarchy (see Figs. 2 and 3). The suggested instances (*claim handling*, *receive claim*, etc.) have been also used to create the optional insurance (sub)domain in the corresponding branch of the hierarchy, as presented in Sect. 4.3.

P2

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

We understand from the way the requirement is written that the set of gateways is fixed and not likely to change. Also, we consider that, semantically speaking, they belong to the same level of abstraction than task, process, etc. This decision is reinforced by the fact that the same gateways are common to all processes. Hence, we choose

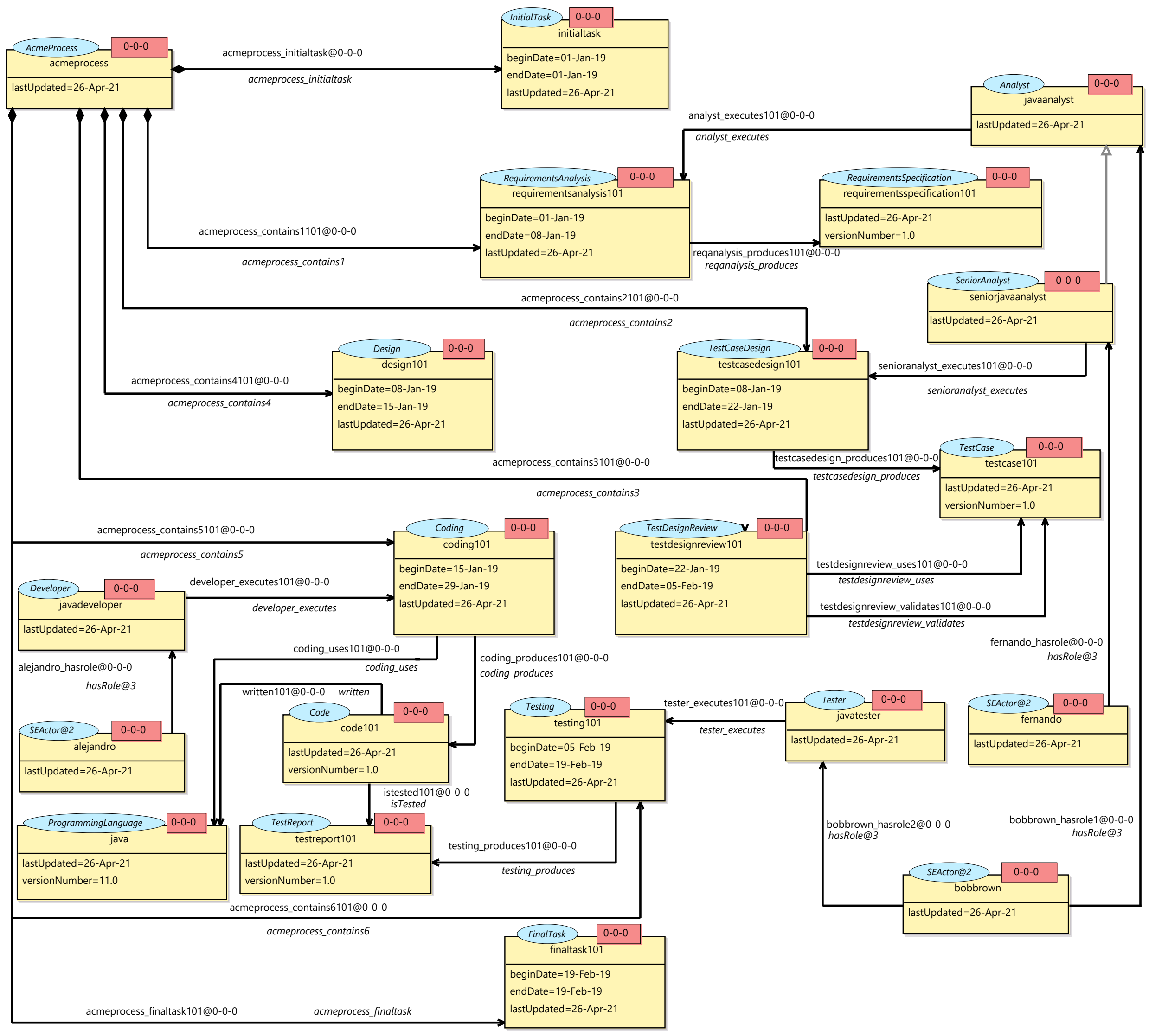

Figure 15: Complete Acme software engineering execution

to define Gateway as an abstract node in the process model, and include the four kinds of gateways as children of it—i. e. related via inheritance, exploiting the fact that MultEcore also allows this kind of relation, as explained in Sect. 4.1. The rationale behind declaring Gateway as abstract is that all processes must use one of its children types for defining sequencing of tasks, but it does not make sense to create an instance of the parent. Finally, it should be noted that, while inheritance is a less flexible construction than typing, adding new kinds of gateways (e. g. xor-split and xor-join) could still be achieved by adding them as new children of Gateway. We refer the reader to Sect. 4.5 for a discussion on how the operational semantics of these gateways could be easily specified with Multilevel Coupled Model Transformations.

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).*'

This requirement is also addressed using inheritance relations in the process model, following a similar rationale as in the previous one. Therefore, we include the nodes InitialTask and FinalTask, and define specialised containment relations from node Process into them, instead of reusing the one for intermediate tasks. More importantly, these two relations define different cardinalities to enforce a unique initial and at least one final task per process, as per the requirement. We do not define inheritance relations among these different containment relations since that kind of construction is not supported in MultEcore.

P4
'*Each task type is created by an actor, who will not necessarily perform it. For example, Ben Boss created the task type assess claim.*'

This requirement entails in our solution the definition of the Actor node and the creates relation from it into Task. The other relation performs hinted in this requirement is discussed in the following one. The example instances mentioned in the requirement are also used in the lower model of the insurance branch of the hierarchy (see Sect. 4.3).

P5
'*For each task type, one may stipulate a set of actor types whose instances are the only ones that may perform instances of that task type. For example, in the XSure insurance company, only a claim handling manager or a financial officer may authorize payments.*'

First, we include another relation from Actor to Task, called performs. However, this is not enough to model which types of actor can execute which types of tasks. As pointed out before in Sect. 4.1, we split the concept of actor as an actual person (e. g. *Ben Boss*) and as a specific role that a person may play (e. g. *claim handling manager*) to allow for the flexibility of several people being able to play the same role, and also for the same person to perform more than one role. Therefore, apart from the Actor node discussed in the previous requirement, we create the different Role nodes, some of which appear as a response to following requirements. For the purpose of fulfilling this requirement, the way we model the semantics that an actor is allowed to perform a task, is by checking that it has a role which can execute that task. Therefore, we create the aforementioned nodes, plus the hasRole and executes relations, so that the semantics are encoded in the Actor – Role – Task triangle.

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).*'

A naive way to address this requirement could consist of the creation of yet another relation (called assigned or something similar) between Actor and Task. But since we define the Role nodes to fulfil other requirements, we can simply take advantage of the *triangle* mentioned in the previous one, and create a role that is assigned to both actors. In such a way, we cover this requirement without needing to define any new elements.

We argue that, besides being a flexible construction, this way of modelling the requirement makes sense from a semantic point of view: there should be some common ability, permission or status that makes those people suitable to perform the task, and we allow modelling it explicitly. Again, the examples used in the requirement are created as instances on the insurance branch, and we also attach plausible roles to those actors to complete the model.

P7

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

This requirement is tackled by simply defining the Artifact node and the uses and produces relations. Again, the examples mentioned in the requirement are used to construct the optional insurance branch in our solution.

P8

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

We just need to add the expectedDuration attribute of type Integer to the Task node to complete this requirement.

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.*'

Once again, we can take advantage of the separation of actors and roles to avoid creating a child node of Task for critical tasks. Instead, we add a simple Boolean attribute isCritical to Task. Since in our solution the information about what an actor can do is not stored in Actor itself but in Role, we create a child node of the latter called SeniorRole. That is, the actor which performs a task marked as critical, must have at least one senior role, which must be able to execute such task, as indicated by the executes relation. In this case, we do not use an attribute to distinguish roles from senior roles since we later use combined roles through a composite pattern, which is more easily illustrated using inheritance relations. With these elements, we can create a similar constraint to the one defined in Sect. 4.4 to ensure that critical tasks are only performed by senior actors. This constraint is shown in Fig. 16, and it states that if any role (hence r:AbstractRole) is connected to a critical task, that role must be a senior one (therefore r:SeniorRole in the TO block).

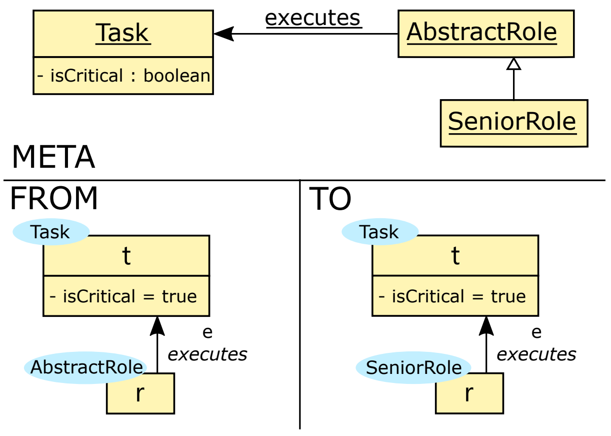

Figure 16: Constraint satisfying part of requirement P9 *related to senior roles and critical tasks*

Regarding the fact that '*the artifacts they produce must be associated with a validation task*', we used the concept of validation tasks for the software branch only. We think that creating an specific child node of Task in process for validation tasks is not a good alternative in this case, as it would pollute that model with software-specific concepts—other kinds of processes may not have validation tasks. If validation tasks were required in the insurance branch—or any other potential new branch—the concept could be easily lifted up. Hence, we refine Task into SEValidationTask in the software process model in level 2, and create a validates edge from it into SEArtifact. Then, in the Acme software process model in level 3, we define TestDesignReview:SEValidationTask (instead of being of type Task as the others), and instantiate validates to indicate that it is used for that purpose for TestCase. We also instantiate uses between both nodes, even if it could be considered redundant in this case, for the sake of coherence and completeness. To enforce the fact that the artefacts created by critical tasks must be validated, we define another MCMT constraint, depicted in Fig. 17. In this constraint, any sea:SEArtifact that is produced by a critical task, must be connected to an vt:SEValidationTask.

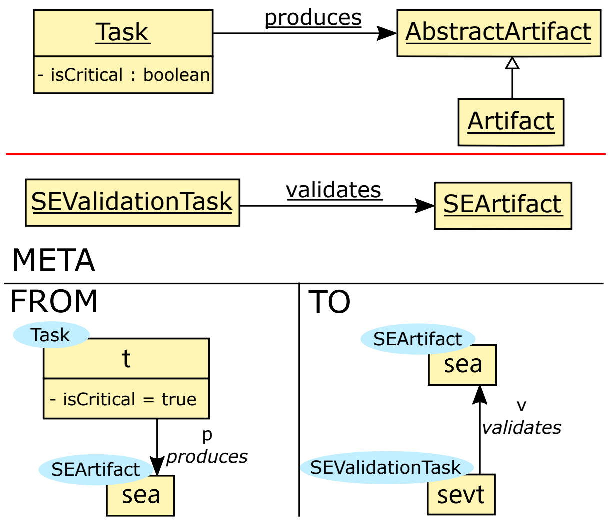

Figure 17: Constraint satisfying part of requirement P9 *related to validation tasks*

P10

'*Each process type may be enacted multiple times.*'

This requirement is trivially addressed in our solution since we allow for multiple instantiations of a process, as our hierarchy shows.

P11

'*Each process comprises one or more tasks.*'

The contains relation specified for P1 from Process to Task, and the way they are instantiated, already covers this requirement.

P12

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

Both attributes have been declared in Task, and are instantiated in the corresponding node, at the lower model of the insurance branch in our hierarchy.

P13

'*Tasks are associated with artifacts used and produced, along with performing actors.*'

This requirement is addressed by creating two new relations in the process model: uses and produces from Task to AbstractArtifact. The performs relation that we discuss in earlier requirements is also used to satisfy this one.

P14

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

Thanks to the way we model the structure of this elements in the process model, this requirement is trivially solved by instantiating Artifact, Task and the uses and produces relations between them appropriately. We show how this can be achieved in the example instance models at the bottom of both branches in our hierarchy.

P15

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

Thanks to the separation of actors and their roles in our solution, this requirement can be easily addressed. The hasRole relation has a 0..* cardinality, so an actor can have several roles by default. But in order to improve the reusability of roles, we choose to include the concept of CombinedRole, which realises the composite pattern (Gamma et al. 1994). In such a way, a combination of roles that several actors share can be defined just once as an instance of CombinedRole and related to several actors. Apart from the aforementioned advantages, using this construction we also remove the need for multiple typing, which is commonly not supported in MLM formalisations, like those based in Set Theory (Kühne and Schreiber 2007) or Graph Theory (Rossini et al. 2014), including our own formalisation (Wolter et al. 2019). Some authors even argue that multiple ontological types are not desirable from an ontological point of view (Atkinson and Kühne 2002). Using MultEcore's supplementary hierarchies (see Sect. 2.2) as a means to add additional types did not make sense in this context in any case, since there are no differentiated domains that justify such a construction.

P16

'*Likewise, an artifact may have more than one artifact type.*'

Same as we do for roles, we use a composite pattern to address this requirement. In such a way, instead of using multiple typing (which, as argued before, is neither desirable nor a possible alternative in MULTECORE), we can join instances of a simple Artifact into an instance of a CombinedArtifact, and use the latter as a replacement of multiple typing.

P17

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

Once again, the triangle construction of actors, tasks and roles allows us to fulfil this requirement without adding any new elements to the models, but defining a constraint over them. First, we explicitly represent in our models that an actor performs a task and also has a series of roles, some of which are allowed to execute certain types of tasks. Then, when instances of Role and Task are created in lower levels, it can be checked that they instantiate the corresponding relations in order to verify this requirement, using the constraint from Fig. 11.

P18

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

The nature of inheritance in MULTECORE allows us to easily model this requirement. Since we use distinction between actors and roles, this requirement actually affects the latter in our solution, according to our understanding: a role can specialise another role, but it does not make sense for an actual person to inherit from another in this context. That is, an actor can have a role, and a separate actor a second role which inherits from the first. In such a way, the specialising role (child) would inherit its executes relation to a task from the specialised role (parent), and any actor having the specialising role would be allowed to perform that task too. However, since this requirement also mentions the possibility of 'senior' versions of the different roles (initially mentioned in P9), we also include a specialisation of Role into SeniorRole, which can be directly instantiated in order to recognise those specialised roles involving seniority. The consequence here is that an X:Role can be specialised into a Y:SeniorRole, which is allowed by MULTECORE. In general, a node can inherit from another as long as their potencies match and their types are the same, or alternatively the specialising node's type is itself a specialisation of the specialised node's type. This is required by our formalisation to ensure that the upwards typing chains (i. e. the sequence of the type, type of type, etc. of an element) are consistent throughout the hierarchy and do not violate the internal constraints of MULTECORE. In any case, we do not think this is an undesirable constraint even from a practical point of view, since it seems straightforward that the relation between a node and its parent have to be close, both structurally and semantically, for inheritance to be established. An excerpt of our models illustrating this scenario is shown in Fig. 18. Finally, the constraint that enforces that only actors with the right role might execute a particular task also ensures that this requirement is fulfilled (Fig. 11).

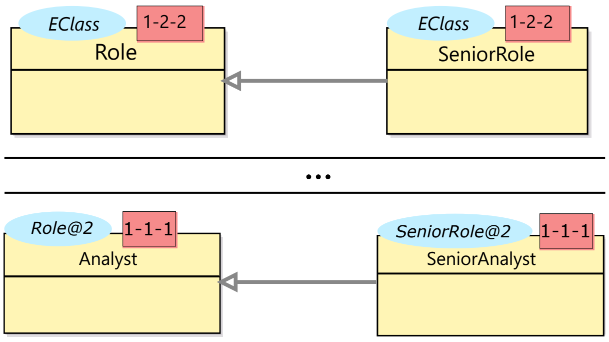

Figure 18: Fragment of the process multi-level hierarchy showing the P18 requirement fulfilment

P19

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

The key to fulfilling this requirement is defining a lastUpdated attribute with the loosest potency possible (1-*) in such a way that it can be instantiated no matter how the hierarchy grows—be it in depth, in width or in number of elements in a model—without forcing the modeller to add neither more definitions of that same attribute, nor typing or inheritance relations to previously defined elements, nor any other mechanism that entails accidental complexity (Atkinson and Kühne 2008). With this goal in mind, we considered four alternative constructions that were possible in our solution. First, we could naively add a copy of the attribute to every node without a parent in the process model. But this solution would forbid the actual elements in that model from instantiating the attribute, so it is not a perfectly valid option. Second, we could add an extra model on top of process (displacing all models in the hierarchy one level lower) for the definition of a single node TimeStamp which contains the aforementioned attribute definition, and type every node in the process model by it. This alternative would give us the desired effect, but does not make any sense semantically. Besides, it is an ad-hoc solution which could cause trouble if we eventually need that level on top for other purposes. The last two options are based on the use of our supplementary typing mechanism to separate concerns, since we can consider time-stamping an aspect that could be included in many domains without being an integral part of any of them. In such a way, we can define a supplementary hierarchy with a single model (two, if we count Ecore on level 0), which contains the TimeStamp node with the lastUpdated attribute with 1-1 potency and 1..1 multiplicity, since the requirement states that nodes *must* instantiate it. So, the third option we considered consisted of adding this TimeStamp as a supplementary type to every other node in the application hierarchy. However, this option is far from ideal, since every new node that we add to the hierarchy needs to be double-typed with this supplementary type to be able to instantiate the attribute. So, finally, the fourth option which we actually implement in our solution is an improvement of the previous one: we change the attribute potency to 1-* and only add TimeStamp as a supplementary type to all nodes in model process. With this construction, we only define the attribute once, 'link' it less times (through supplementary typing) and it is already available everywhere thanks to potency in the rest of the hierarchy. Moreover, it would still be available in any new branches, any new models in the existing branches and any new node that we define in the existing models using the types of process. It would only be required to add TimeStamp as a supplementary type by hand if we were to instantiate EClass. So we believe that this is a nearly-optimal solution for this requirement.

To sum up the discussion of the P*X* requirements, Tab. 1 summarises whether they have been tackled in our solution and how.

Secondly in this section, we discuss the requirements which are specific for software engineering processes (Sect. 2.3 in the challenge description). We begin by reproducing in MultEcore the diagram shown in Fig. 1 in the description, both of which are included in our Fig. 5 for a side-by-side comparison.[4] Using this figure as a starting point, we add the different nodes and edges that we require to fulfil those requirements. The full model is depicted in Fig. 6. Again, we refer to them with their original names of the form S*X*, and summarise the following discussion at the end of this section, in Tab. 2.

S1

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

RequirementsAnalysis is present in the depiction of the Acme software engineering process

[4] Note that each instance of Sequence is depicted as a node, plus the two arrows which indicate its source and target tasks.

Table 1: Summary of PX requirements

Req.	Addressed?	Comments
P1	+	—
P2	+	Using inheritance
P3	+	Using inheritance
P4	+	—
P5	+	Modelled as a triangle between the nodes Actor, Role and Task
P6	+	Does not require new modelling constructs
P7	+	—
P8	+	—
P9	+	Using two constraints
P10	+	Trivially addressed
P11	+	Addressed in P1
P12	+	—
P13	+	Partially addressed earlier
P14	+	Trivially addressed
P15	+	Addressed with composite pattern
P16	+	Addressed with composite pattern
P17	+	Using a constraint
P18	+	Using inheritance and an existing constraint
P19	+	Using a supplementary hierarchy

challenge description and is therefore included already in the initial version of the model in Fig. 5. To that same model—acme software engineering process, which we refer to as just acme process in the remainder—we add the nodes Analyst:Role and RequirementsSpecification:SEArtifact, plus the corresponding relations, according to the process model (Fig. 3). The usage of SEArtifact instead of Artifact as type in the latter node is due to requirement S10, and we refer the reader to that discussion for a justification of this choice. This same remark also applies to some of the following requirements.

S2

'*A test case design is performed only by senior analysts and produces test cases.*'

We have simplified the wording of this requirement while maintaining its meaning. To fulfil it, we include SeniorAnalyst:SeniorRole and the corresponding instance of the executes relation in model acme process. We indicate that SeniorAnalyst is a specialised version of Analyst through an inheritance relation, as an example of the construction discussed in P18. We also include TestCase:SEArtifact and instantiate the produces relation to indicate that it is a product of test case design. Note that the senior analyst role does not need to be connected to an actor for this model to be correct. Hence, we choose not to overload the models with additional details beyond the ones enforced by the requirements, in order to simplify their description and visualisation in this paper.

We should also point out that the fact that TestCaseDesign is marked as critical will ensure that the constraint in Fig. 16 (from P9) enforces that the role executing the task is of senior type.

S3

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

To address this requirement we add to acme process the nodes Developer:Role, and two instances of SEArtifact: Code and ProgrammingLanguage. We connect these nodes to the coding task by instantiating, respectively, the relations executes, produces and uses.

S4

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

In order to represent that a code is written in a programming language, we create a relation written between these two artifacts. Since this relation only covers two software-specific artifacts, it does not have a type in the process model. For such scenarios, MultEcore always allows to create direct instances of an EClass or EReference through potency, and in this case we use the latter as the type of written. Although this construction differs conceptually from linguistic extensions (Atkinson and Kühne 2001), its practical usage is quite similar to it.

S5 and S6

'*Coding in COBOL always produces COBOL code.*' '*All COBOL code is written in COBOL.*'

We group together these two requirements since they pertain to the same part of the model and have common elements. Coding in COBOL, as an instance of the Coding task, belongs naturally in a level below the model acme process, since the latter deals with coding as a generic concept, as the original figure in the challenge description shows. Therefore, the new node CodingCOBOL:Coding is declared in the bottom-most model of the software branch of our hierarchy: acme software engineering process configuration, which we call acme configuration for short and is depicted in Fig. 7. This same reasoning can be applied to COBOLCode:Code and COBOL:ProgrammingLanguage. To complete the model, the relevant instances of the relations coding_uses, coding_produces and written are used to connect those three nodes to each other, modelling the semantics of both requirements.

S7

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

The fulfilment of this requirement implies creating an instance of Actor in the acme configuration model. Due to the refinement of Actor (from process model) into SEActor (from software engineering process model, called software process for short) that S10 entails, the node AnnSmith that we create is an instance of SEActor. Note that our implementation allows us to create instances of SEActor in both levels 3 and 4 of the software branch of the hierarchy. Hence, we include AnnSmith:SEActor@2 in the bottom model, and instantiate the performs relation from process (which relates actors to tasks) to indicate that she carries out the task coding in COBOL. As already explained, our solution distinguishes between actors (as actual people) and the roles they perform, so we also create a special type of developer that is allowed to code in COBOL, i. e. COBOLDeveloper:Developer. Ann Smith is connected to this role via an instantiation of the hasRole relation. Finally, even though there is already an instantiation of executes between Developer and Coding in the acme process model, we think that it is appropriate to instantiate it again in this model, between COBOLDeveloper and CodingCOBOL. We believe that this repetition provides clarity to the model and simplifies the definition of the constraint presented in 4.4.

S8

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

The node Testing is already present, so we add the nodes Tester:Role@2 and TestReport:SEArtifact to the model acme process. We instantiate the relations executes and produces (from two levels above) in order to connect each node to Testing, respectively.

S9

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

At first glance, it could be argued that this requirement can be satisfied in the model software process in level 2 of the software branch. However, we believe that it actually belongs to model acme process, since it is only related to Testing, which is declared on that model. Moreover, Testing may not be defined or used in the same way in different software processes which could be defined in other hypothetical software companies. With this choice, we also avoid the need for a constraint that would check that Testing is associated with only some specific instances of SEArtifact. Therefore, we include in model acme process a node TestReport:SEArtifact that is connected to the existing nodes Testing and Code via two relations isTested:EReference and testing_produces:produces@2, respectively. As explained in S4, we exploit the fact that MultEcore allows creating direct instances of EReference anywhere for the typing of isTested. We choose to only create isTested for Code, but there is no obstacle if one wants to create more relations like it from other instances of SEArtifact to other test reports—or even the same one, if one wanted to model a test report that contains info about several tested artefacts.

S10

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

We hinted in the discussion of previous requirements that this one entails, to our understanding, the creation of the intermediate model software process for the software branch, that does not have a counterpart in the insurance branch. In this new model, we need to refine generic artefacts into software engineering artefacts which contain more information. Hence, in the model in level 2 we create Artifact:SEArtifact, which defines the required attribute versionNumber of type String that should be instantiated in the bottom-most level of the branch—i. e. model acme configuration in level 4. The potency that we require for the attribute is therefore 2-2 (recall that the depth for attributes is always 1 and consequently not displayed). The cardinality of this attribute, not displayed, is 1..1, so that the attribute *must* be instantiated, according to the requirement. In such a way, any X:Y:SEArtifact in model acme configuration needs to instantiate the attribute, as COBOL and COBOLCode illustrate. To fulfil the rest of the requirement, we also need to model that instances of SEArtifact have a special relation to actors. Since MultEcore does not allow for cross-level relations, we need to create a corresponding SEActor:Actor in software process so that we can then define responsibleActor:EReference among them. Using the same rationale as for the attribute, the potency of responsibleActor is 2-2-1 and its cardinality is 1..1. Examples of instances of this relation are those connecting COBOL and COBOLCode to JohnDoe in model acme configuration.

S11

‘*Bob Brown is an analyst and tester. He has created all task types in this software development process.*’

We interpret that ‘*this software development process*’ refers to a specific instance of a software process. That is, the model in Fig. 1 in the challenge description, which corresponds to model acme process in our solution. Hence, we include in that model a node BobBrown:SEActor and instantiate the creates relation from it towards every instance of Task in this model, e. g. Design. This includes the initial and final tasks that every process must have. It is worth pointing out again that our solution allows for the creation of direct instances of actors in two different levels, both in the insurance branch (as instances of Actor) and in the software branch (instantiating SEActor). This construction is necessary since the two lower levels in both branches of our hierarchy (levels 2 and 3 on insurance branch; 3 and 4 in software) may need to define actors in order to adhere to the requirements, e. g. Bob Brown needs to appear in model acme process and Ann Smith in acme configuration. In contrast, roles can be simply instantiated and re-instantiated in those levels, since the domain naturally requires so, e. g. COBOLDeveloper:Developer:Role@2. We have made this design choice to allow for the representation of specific roles, like a COBOL developer, without losing the information that such a role is also in the category of developers, in general.

While this construction for actors might seem undesirable at first, we argue that it removes the need for cross-level relations and that it neither requires any additional elements to be defined nor enforces an artificial re-instantiation of actors—which would be done in a similar manner as we do for roles. The only shortcoming that we see in our solution is that the same actor may appear twice in two models in adjacent levels. For example, if Ann Smith would be responsible for creating tasks that appear in acme process, she would also have to appear there along Bob Brown, and hence would be a duplicate of the Ann Smith that is already present in acme configuration. However, if some practical application of our models—like code generation—were affected by such duplication, we could simply identify both nodes based on the fact that they share the same name, type and potency.

S12

‘*The expected duration of testing is 9 days.*’

Testing in model acme process instantiates the expectedDuration integer attribute to 9 to fulfil this requirement.

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.*’

We instantiate the isCritical Boolean attribute to true in TestCaseDesign, in model acme process. We connect the node SeniorAnalyst to that instance of Task with an instance of executes. For the sake of simplicity, we do not relate this role to any actor, although it would be reasonable to do so eventually. The fact that TestDesignReview validates TestCaseDesign is represented by TestDesignReview being of type SEValidationTask and instantiating the validates relation towards every artefact produced by TestCaseDesign, namely TestCase. The constraint in Fig. 17 (from P9) ensures that this construction is enforced.

6 Assessment of the Modelling Solution

In this section, we discuss the advantages and shortcomings of the choices we made in our solution to the challenge. We also point out whether we were forced to make any compromises or whether our solution presents any deficiencies.

6.1 Basic modelling constructs

MultEcore is graph-based from a theoretical point of view, and this fact reflects on the EMF-based implementation. All models use nodes and relations as the basic building blocks, which are contained in models. Attributes are formally nodes, as explained in Macías (2019), but in practise they behave as commonly expected: they are defined inside a node and instantiated in the instances of that node. The rationale for the separation of these elements in different models is to make them as independent from each other as possible, so that they can be connected to each other only by typing relations. This provides an advantage when adding and removing intermediate models. For example, software process could be removed from our hierarchy, and the types of the elements in the models below just be replaced by the type of the removed types, e. g. SEActor to Actor and responsibleActor to EReference. To achieve this separation, potency plays an important role, as discussed later in this section. Furthermore, combining inheritance with typing also allows us to choose whichever construction is more flexible, understandable and aligned with the requirements, e. g. the composite pattern that we present for roles.

6.2 Levels

Levels are used as an organisational tool in MultEcore, as explained in Sect. 2.1. This rationale entails that the typing relations—from nodes to nodes and from relations to relations—have the meaning of *my type defines my structure*, in the sense of which relations can be defined and to

Table 2: Summary of SX requirements

Req.	Addressed?	Comments
S1	+	—
S2	+	Enforced by constraint from P9
S3	+	—
S4	+	Creating a direct instance of EReference through potency
S5	+	Addressed in new model acme configuration
S6	+	Addressed in new model acme configuration
S7	+	Using roles
S8	+	—
S9	+	Addressed in acme process, not in software process
S10	+	Addressed in new model software process
S11	+	May entail actor duplication
S12	+	—
S13	+	—

which other nodes, which attributes can be instantiated, which nodes can inherit from which other nodes, etc. Due to potency, these typing relations can jump over levels, but still levels serve as a default organisation of models and the elements they contain. We also mentioned already in Sect. 2.1 that these typing relations among levels do not necessarily adhere to classification with all its implications, since we prioritise flexibility and conciseness, but are in general quite aligned with the concept.

6.3 Number of levels

As stated before, hierarchies in MultEcore are unbounded, so the hierarchy we present could grow downwards as much as necessary. We chose to add an intermediate level for refinements related to software processes (e. g. SEActor), which could perhaps have been done with inheritance in model process. However, this alternative would pollute the model, which is supposed to be generic and unaffected by the particularities of any subdomain. Conversely, we did not force a similar intermediate level in the insurance branch just to keep the hierarchy symmetric since it was not necessary, but of course it could be included if required at a later point in time. To sum up, we designed our solution to be as flexible as possible, and used levels to create, from our perspective, clearly-defined partitions of the domain: processes in general, software processes, the software process of a particular company, and the state of such process at a specific point in time (and a similar partition for the insurance subdomain).

Actually, any of the models in the intermediate levels can be considered a DSML which is used to define the level(s) below it, using the types they define in a structurally coherent manner and satisfying the given constraints. The bottom-most models represent a specific state of the process, e. g. *Ann Smith, who is a COBOL Developer, is using COBOL version 1.3 to implement version 3.1 of a particular piece of COBOL code*. These bottom-most models could be used for different purposes, like logging the different tasks performed by the actors and the generated artefacts, or for monitoring purposes, by representing the current state of the process. If the models were enhanced with further details, one could even consider the execution of simulations prior to the actual enactment of the process in the real world. In such a way, it would be possible to assess whether the specified process, task distribution, workload, etc. are likely to succeed or will probably lead to time and budget overruns.

6.4 Cross-level relationships

Cross-level relations go against modularity, and therefore would reduce some benefits of our approach, e. g. flexibility and reusability. Moreover, they are purposely not supported by our current formalisation (Wolter et al. 2019). Hence, our solution does not employ them, but we have not identified any case in which they are more desirable than an alternative construction.

6.5 Cross-level constraints

The expressive power of MCMTs allows us to use them to define different kinds of semantics, which have been illustrated in this paper. We can specify dynamic semantics to describe the behavioural aspect of the modelled system and also define static semantics that check the structural correctness of the multi-level hierarchy. As discussed in Sect. 4.5, the dynamic semantics are applied following the traditional in-place model transformations rules manner where the match of the left-hand side of the rule leads to the modifications specified on the right-hand side. The cross-level constraints would be executed in a so-called *check mode* where the left-hand side and the right-hand side specify two multi-level sub-hierarchy patterns that have to be found for the constraint to be satisfied (see Sect. 4.4).

6.6 Integrity mechanisms

This discussion is twofold: integrity mechanisms which prevent incorrect constructions and repairing mechanisms if such a construction is made. For the first group, both the formalisation and the implementation of MultEcore has mechanisms to avoid cyclic inheritance, cyclic typing, potency-violating typing, invalid inheritance, multiplicity violations for relations and attributes, duplication of elements and incorrect typing relations for all kinds of elements. Repairing actions like the ones required for the co-evolution of models, metamodels and MTs are not part of MultEcore. However, the tool does include some basic repairing mechanisms, e. g. fixing the potency of an element to `0-0-0` if any of the three values becomes `0` or correcting depth of an element to the depth of its type minus one, if a higher or equal value is specified. Additionally, more advanced repair mechanisms are planned in future releases, such as changing the type of an element to the type of its type if the former is removed.

6.7 Deep characterisation

Our solution makes intensive use of potency, with no element using MultEcore's default potency of `1-1-*`, except for the supplementary node `TimeStamp`. The reasons for this are twofold. First, the presented scenario clearly defines a bottom-most level for model instances (i. e. enactments) of specific processes, and it does not make sense to create further instances of such models. Hence, the value of depth is always bounded. And second, the way in which most elements in the top models, especially `process`, are expected to be used, forces us to use end values higher than `1`. In some cases, even the start value differs from `1` to prevent them from being instantiated in the level below, e. g. the `performs` relation in `process`.

6.8 Generality

We believe that our solution performs very well regarding its generality, and the reusability that it entails. We have managed to create a solution with minimal redundancy in most cases, the only exceptions being the potential duplication of actor in two adjacent levels and the several supplementary typing relations in `process` to enable the instantiation of the attribute `lastUpdated` in all nodes. Moreover, we illustrated the reusability of the `process` model and the related MCMT rules by modelling the optional insurance domain, and including an example execution of this process in our solution. In general, we believe that the `software process` model can be used for other software-related companies that may implement different processes than Acme. Likewise, the `acme process` model one level below could both be instantiated for other points in time of the same enactment (as illustrated by our MCMT-based execution) or enacted differently for other departments of the same company that adhere to the same process.

6.9 Extensibility

Our solution already illustrates how some extensions can be performed when new requirements appear. For example, the discussions regarding senior actors and validation tasks in P9. Similar extensions through inheritance are always available and simple to perform, since they only add new information to the models, therefore not compromising their integrity or semantics. Moreover, we have shown how model (or level) insertion can be performed by introducing an intermediate level in the software branch (motivated by S11) which has no counterpart in the insurance branch of our hierarchy.

We finalise this section by discussing two of the topics that are recommended in the challenge description.

First, we would like to remark that MultEcore is not only the supporting tool for our approach that we have used to fully create the models and MTs presented in this challenge. The approach also includes a detailed formalisation based on Graph Theory and Category Theory which provides a framework of reference for the tool's behaviour (Wolter et al. 2019).

And second, although we already stated that the verification of our models can be performed through integrity mechanisms (Sect. 6.6), there are additional checks. For example, the standard validators of EMF for Ecore models and their XMI instances can be still used with MultEcore, since the tool reflects multi-level changes on both facets of the models in each level, using a mechanism called *sliding window* presented in Macías (2019, Sect. 4.1). This validator can be used to check that obligatory attributes are correctly instantiated or that the multiplicities of relations are respected, among others. However, these checks have some caveats due to the way in which multi-level aspects are represented in those models, so a full integration that does not display multi-level constructions as errors is still a matter of future work.

7 Related Work

The MULTI challenge has received several responses by the community in order to bring insights on how MLM can be applied to solve the suggested scenarios. We first discuss other solutions related to the Process Challenge in 2019 (Almeida et al. 2019).

Jeusfeld's solution (see Jeusfeld (2019b)) is implemented in DEEPTELOS (Jeusfeld 2019a; Jeusfeld and Neumayr 2016) that extends TELOS and that allows hierarchies of level objects (called *most-general* instances) to be defined. DEEPTELOS is developed by just creating the DEEPTELOS objects with additional rules/constraints in ConceptBase (Jarke et al. 1995). Note that the core idea of DEEPTELOS is to exploit the powertype pattern (Odell 1994) and therefore is a level-blind approach (Henderson-Sellers et al. 2013), which means that it does not express an explicit notion of level, even though they are intuitively derived by analysing the solution implementation. This powertype-based solution allows them to naturally deal with cross-level relationships, feature that we do not support in MultEcore. On the other hand, Jeusfeld argue that certain requirements, such as P17 cannot be completely fulfilled as they would have to extend their specification by TELOS rules. Conversely, our multi-level transformation language (MCMTs) allows us to specify multi-level constraints. The most-general instances idea replaces the well-known potency mechanism present in level-adjuvant approaches. In particular, our three-value potency specification allows us to be both generic and precise depending on the particular needs. It is unclear to us how such a level of precision could be achieved using the solution in Jeusfeld (2019b), where they also have to make the explicit separation between `Task` and `TaskType`, which we deem unnecessary in a multi-level context.

Somogyi et al. (Somogyi et al. 2019) also contributed with their solution by using their tool DMLA (Theisz et al. 2019; Urbán et al. 2018). DMLA is a self-validating metamodelling formalism relying on gradual model constraining through its interpretation of the classical instantiation relation. DMLA is self-described, and it also provides so-called *fluid metamodelling*, which means that it is not required to instantiate all entities of a model at once. Models in DMLA are stored in tuples, referencing each other, and thus, forming an entity graph. It is also a level-blind approach that naturally supports the specification of cross-level relationships. Being a level-blind approach where all entities can reference any other entity (the fluid nature), it is easier for the modeller to construct invalid models, which is more difficult in other approaches where the hierarchy of models is clearly constructed, like ours. Furthermore, the sanity checks facilitated by potency allow the modeller to always be sure that the model under construction is correct. Also, DMLA does not explicitly support some features, such as inheritance (even though the authors argue that it can be simulated). While MultEcore naturally supports inheritance, Somogyi et al. had to simulate inheritance which resulted in an artificial workaround to solve some of the requirements.

The two solutions discussed above were the only ones published along with our MultEcore response in 2019. However, in 2018 there has been another challenge case, namely the Bicycle Challenge[5] , it is interesting to discuss the work presented by Lange and Atkinson (Lange and Atkinson 2018). Note that Mezei et al. (Mezei et al. 2018) also presented a solution using DMLA, for which we do not enter into more details as the relevant aspects have already been discussed in the previous paragraph.

Lange and Atkinson's solution (Lange and Atkinson 2018) was constructed using the mature tool MELANEE (Atkinson and Gerbig 2016). MELANEE is one of the most advanced tools based in OCA (Atkinson and Kühne 2005) for deep modelling which supports modelling through deep, multi-format, multi-notation, user-defined languages. The MELANEE solution is closer to what our solution with MultEcore looks like as it is a level-adjuvant approach that also distributes models according to the ontological classification of its elements and uses (a different form of) potency. Like in MultEcore, MELANEE does not allow cross-level relationships so models are organised into clear abstraction levels. While this has some advantages, it also has some drawbacks, for instance, the creation of additional nodes in certain levels to make the connections. An example reflected in our solution is the fact that an actor may appear in two different abstraction levels. If we take as a reference the right branch of the multi-level hierarchy depicted in Fig. 2, while an actor can create tasks in level 3 of this branch (see, for example, `BobBrown` on the right side of Fig. 6) it can also perform concrete tasks such as `CodingCOBOL`, performed by `AnnSmith` (see bottom right of Fig. 7).

Finally, regarding our own submission to the challenge in 2019 (see Rodríguez and Macías 2019) we have made improvements and extensions both to the solution and to the MultEcore tool. The multi-level hierarchy presented in the previous work was modelled so it was symmetric, i. e., both branches (insurance and software engineering) had the same length. This forced us to include an intermediate model for the insurance domain that did not really capture any of the requirements stated in the challenge description. In the current version presented in this article this model has been avoided, which helped us demonstrate a flexibility aspect of MultEcore where the different domains do not need to have the same length as they are fully independent from each other. Moreover, as demonstrated, the MCMT rules are still applicable to both domains due to their vertical and horizontal flexibility (for more details on this, we refer the reader to Rodríguez et al. 2019a, Sect. 4.2). The composite pattern was already implemented for roles in the solution submitted in 2019. MultEcore's facilities such as the use of inheritance and the potency customisation capabilities allowed us to exploit such a pattern within the multi-level context. Thus, we have also used this construction to discuss

[5] Bicycle Challenge 2018: https://www.wi-inf.uni-duisburg-essen.de/MULTI2018/wp-content/uploads/2018/03/MULTI2018-BicycleChallenge.pdf

how to model the artefact situation (P16), and the scenario on Fig. 18. Furthermore, we explore in this paper the operational semantics of process challenge. We have shown in Sect. 4.5 how we can execute models and evolve them by applying MCMT rules that describe the behaviour. This part was not examined in our submission in Rodríguez and Macías (2019). We have also included some minor enhancements which we previously overlooked, like the values of some potencies or making the node `Gateway` abstract. Finally, we have improved the way in which supplementary attributes can be instantiated to develop a nearly-optimal solution for requirement P19.

8 Conclusions and Future work

In this paper, we have presented an extended solution based on our initial contribution (Rodríguez and Macías 2019) to the Process Challenge proposed at the MULTI workshop (Almeida et al. 2021). Our multi-level modelling hierarchy has a total of five abstraction levels, two branches and 7 models (more if we take into account all the model states generated during the execution, shown in Sect. 4.5). Such hierarchical distribution covers the generic domain of process description and its refinement for the software engineering and the insurance domains. Each level can be understood as a potential candidate for the generation of software artefacts, like domain-specific editors (graphical and/or textual) to specify processes at any level of abstraction, or for the simulation of processes through model transformations at the bottom levels. Our solution is based on the MultEcore tool and the infrastructure that connects it to maude which allows us to perform simulation/execution. MultEcore is built on top of EMF which allows us to use all the EMF capabilities boosted with multi-level capabilities. For instance, this facilitates the usage of the rich ecosystem of EMF such as using editors with Sirius for graphical results and Xtext for custom specification languages.

From a more conceptual standpoint, one of our ambitions with respect to MultEcore is to make it an approach that enhances flexibility and reusability. This has allowed us to create an elegant, concise and correct multi-level hierarchy for the given domain of process modelling where, for example, the branches are independent and their lengths are different. We believe that this solution can be an interesting contribution for the challenge and be used to foster fruitful discussions within the MLM community. Furthermore, we have gone one step further by exploring behavioural aspects, and we believe that including this dimension as part of future challenge proposals would bring engaging results from the MLM community.

We have presented preliminary results regarding execution by showing some examples of model evolution by applying operational semantics via MCMT rules. Currently, we are actively working on MultEcore–maude infrastructure to improve the execution and further verification of the specified multi-level hierarchies. Also, we are studying how to improve the MCMTs flexibility, by taking advantage of inheritance to reuse some MCMT rules with common behaviour. While MCMTs are flexible with respect to horizontal and vertical extensions, we identify a key point of improvement as being able to reuse `META` levels on MCMTs into other rules. Another important aspect that we plan to work on, is the implementation in maude and the integration into MultEcore of the check mode of MCMTs for the validation of the multi-level hierarchy with respect to structural constraints (as shown in Sect. 4.4).

We conclude this paper by answering the questions that the challenge description explicitly asks respondents to address.

‘*Does the submission address the established domain as described in Sect. 2 and demonstrate the use of multi-level features?*’ We believe that our solution contains all the required concepts and constructions required in the challenge description. In most cases, these constructions do not require workarounds or additional concepts, and we discuss and justify our choices in the few cases where we need them. Furthermore, our solution prominently makes use of multiple levels, three-valued potency specification and double typing (through a supplementary hierarchy). All of these concepts are important multi-level features that this submission showcases.

'*Does it evaluate/discuss the proposed modeling solution against the criteria presented in Sect. 3?*' The whole Sect. 6 in this paper is dedicated precisely to the discussion of those criteria, in the same order in which they are enumerated in Almeida et al. (2019), so that we can make sure that this question is properly addressed. We also included the recommended discussion aspects suggested by the challenge description.

'*Does it discuss the merits and limitations of the applied MLM technique in the context of the challenge? Authors may suggest further requirements that clearly demonstrate the utility of their chosen approach.*' We have thoroughly discussed the advantages of MultEcore and the few scenarios where we found limitations throughout Sects. 2, 5, 6 and 7. We have also suggested including new requirements regarding the operational semantics of the challenge's domain for upcoming editions in Sect. 4.5.

References

Almeida J. P. A., Kühne T., Rutle A., Wimmer M. (2021) The MULTI Process Challenge–EMISAJ Special Issue Version. http://purl.org/emisajchallenge

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

Arendt T., Habel A., Radke H., Taentzer G. (2014) From Core OCL Invariants to Nested Graph Constraints. In: Graph Transformation - 7th International Conf., ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings. Lecture Notes in Computer Science Vol. 8571. Springer, pp. 97–112

Atkinson C., Gerbig R. (2016) Flexible Deep Modeling with Melanee. In: Betz S., Reimer U. (eds.) Modellierung 2016. LNI Vol. 255. Gesellschaft für Informatik, Bonn, pp. 117–122

Atkinson C., Gerbig R., Kühne T. (2014) Comparing multi-level modeling approaches. In: Proceedings of the Workshop on Multi-Level Modelling co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014. CEUR Workshop Proceedings Vol. 1286. CEUR-WS.org, pp. 53–61

Atkinson C., Gerbig R., Tunjic C. (2012) Towards Multi-level Aware Model Transformations. In: Theory and Practice of Model Transformations - 5th Intl. Conf., ICMT 2012. Lecture Notes in Computer Science Vol. 7307. Springer, pp. 208–223

Atkinson C., Gerbig R., Tunjic C. V. (2015) Enhancing classic transformation languages to support multi-level modeling. In: Software & Systems Modeling 14(2), pp. 645–666

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

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

Atkinson C., Kühne T. (2002) Rearchitecting the UML infrastructure. In: ACM Transactions on Modeling and Computer Simulation (TOMACS) 12(4), pp. 290–321

Atkinson C., Kühne T. (2005) Concepts for Comparing Modeling Tool Architectures. In: Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings.

Lecture Notes in Computer Science Vol. 3713. Springer, pp. 398–413

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

Clark T., Warmer J. (2003) Object Modeling With the OCL: The Rationale Behind the Object Constraint Language Vol. 2263. Springer

Clavel M., Durán F., Eker S., Lincoln P., Martí-Oliet N., Meseguer J., Talcott C. L. (eds.) All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Lecture Notes in Computer Science Vol. 4350. Springer

Durán F., Eker S., Escobar S., Martí-Oliet N., Meseguer J., Rubio R., Talcott C. L. (2020) Programming and symbolic computation in Maude. In: J. Log. Algebraic Methods Program. 110

Durán F., Garavel H. (2019) The Rewrite Engines Competitions: A REctrospective. In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Proceedings, Part III. Lecture Notes in Computer Science Vol. 11429. Springer, pp. 93–100

Durán F., Rodríguez A. (2021) Towards a Maude-based implementation of MultEcore multilevel modelling languages. In: Actas de las XX Jornadas de Programación y Lenguajes (PROLE 2021. SISTEDES http://hdl.handle.net/11705/PROLE/2021/020

Ehrig H., Ehrig K., Habel A., Pennemann K. (2006) Theory of Constraints and Application Conditions: From Graphs to High-Level Structures. In: Fundam. Informaticae 74(1), pp. 135–166

Gamma E., Helm R., Johnson R., Vlissides J. (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional

Garavel H., Tabikh M., Arrada I. (2018) Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages - The 4th Rewrite Engines Competition. In: Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Proceedings. Lecture Notes in Computer Science Vol. 11152. Springer, pp. 1–25

Habel A., Pennemann K. (2009) Correctness of high-level transformation systems relative to nested conditions. In: Math. Struct. Comput. Sci. 19(2), pp. 245–296

Henderson-Sellers B., Clark T., Gonzalez-Perez C. (2013) On the Search for a Level-Agnostic Modelling Language. In: Advanced Information Systems Engineering - 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings. Lecture Notes in Computer Science Vol. 7908. Springer, pp. 240–255

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

Jeusfeld M. A. (2019a) DeepTelos Demonstration. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 98–102

Jeusfeld M. A. (2019b) DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 66–77

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level Modeling with Most General Instances. In: Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings. Lecture Notes in Computer Science Vol. 9974, pp. 198–211

Kühne T. (2018a) A story of levels. In: Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS 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. 673–682

Kühne T. (2018b) Exploring Potency. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. ACM, pp. 2–12

Kühne T., Schreiber D. (2007) Can programming be liberated from the two-level style: multi-level programming with DeepJava. In: ACM SIGPLAN Notices 42(10), pp. 229–244

Lange A., Atkinson C. (2018) Multi-level modeling with MELANEE. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 653–662

de Lara J., Guerra E. (July 2010) Deep metamodelling with MetaDepth. In: Objects, Models, Components, Patterns. Lecture Notes in Computer Science Vol. 6141. Springer, pp. 1–20

de Lara J., Guerra E. (2018) Refactoring Multi-Level Models. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 27(4), p. 17

de Lara J., Guerra E., Kienzle J., Hattab Y. (2018) Facet-oriented modelling: open objects for model-driven engineering. In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018. ACM, pp. 147–159

de Lara J., Guerra E., Sánchez Cuadrado J. (2015) Model-driven engineering with domain-specific meta-modelling languages. In: Software & Systems Modeling 14(1), pp. 429–459

Macías F. (2019) Multilevel modelling and domain-specific languages. PhD thesis, Western Norway University of Applied Sciences and University of Oslo

Macías F., Rutle A., Stolz V. (2016) MultEcore: Combining the Best of Fixed-Level and Multilevel Metamodelling.. In: Proceedings of the 3rd International Workshop on Multi-Level Modelling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016). CEUR Workshop Proceedings. CEUR-WS.org, pp. 66–75

Macías F., Rutle A., Stolz V. (2017) Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge. In: Proceedings of MODELS 2017 Satellite Event: Workshops. CEUR Workshop Proceedings Vol. 2019, pp. 269–273

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, 10:1–35

Macías F., Wolter U., Rutle A., Durán F., Rodriguez-Echeverría R. (2019) Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. In: Journal of Logical and Algebraic Methods in Programming

Méndez-Acuña D., Galindo J. A., Degueule T., Combemale B., Baudry B. (2016) Leveraging Software Product Lines Engineering in the development of external DSLs: A systematic literature review. In: Computer Languages, Systems & Structures 46, pp. 206–235

Mens T., Gorp P. V. (2006) A Taxonomy of Model Transformation. In: Electron. Notes Theor. Comput. Sci. 152, pp. 125–142

Meseguer J. (1992) Conditioned Rewriting Logic as a United Model of Concurrency. In: Theor. Comput. Sci. 96(1), pp. 73–155

Mezei G., Theisz Z., Urbán D., Bácsi S. (2018) The bicycle challenge in DMLA, where validation means correct modeling. In: Hebig R., Berger T. (eds.) Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 643–652

Odell J. (1994) Power Types. In: J. Object Oriented Program. 7(2), pp. 8–12

Orejas F., Ehrig H., Prange U. (2010) Reasoning with graph constraints. In: Formal Aspects Comput. 22(3-4), pp. 385–422

Radke H., Arendt T., Becker J. S., Habel A., Taentzer G. (2018) Translating essential OCL invariants to nested graph constraints for generating instances of meta-models. In: Sci. Comput. Program. 152, pp. 38–62

Rodríguez A., Durán F., Rutle A., Kristensen L. M. (2019a) Executing Multilevel Domain-Specific Models in Maude. In: Journal of Object Technology 18(2), 4:1–21

Rodríguez A., Macías F. (2019) Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 152–163

Rodríguez A., Rutle A., Durán F., Kristensen L. M., Macías F. (2018) Multilevel modelling of coloured petri nets. In: Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018. CEUR Workshop Proceedings Vol. 2245. CEUR-WS.org, pp. 663–672

Rodríguez A., Rutle A., Kristensen L. M., Durán F. (2019b) A Foundation for the Composition of Multilevel Domain-Specific Languages. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 88–97

Rossini A., de Lara J., Guerra E., Rutle A., Wolter U. (2014) A formalisation of deep metamodelling. In: Formal Aspects of Computing 26(6), pp. 1115–1152

Somogyi F. A., Mezei G., Urbán D., Theisz Z., Bácsi S., Palatinszky D. (2019) Multi-level Modeling with DMLA - A Contribution to the MULTI Process Challenge. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 119–127

Steinberg D., Budinsky F., Merks E., Paternostro M. (2008) EMF: Eclipse Modeling Framework. Pearson Education

Theisz Z., Bácsi S., Mezei G., Somogyi F. A., Palatinszky D. (2019) By Multi-layer to Multi-level Modeling. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C). IEEE, pp. 134–141

Urbán D., Theisz Z., Mezei G. (2018) Self-describing Operations for Multi-level Meta-modeling. In: Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018. SciTePress, pp. 519–527

Wolter U., Macías F., Rutle A. (Nov. 2019) The Category of Typing Chains as a Foundation of Multilevel Typed Model Transformations. 2019-417. University of Bergen, Department of Informatics