Multilevel Modelling with MultEcore: A contribution to the MULTI 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 Challenge is intended to encourage the Multilevel Modelling research community to submit solutions to the same, well described problem. This year the subject domain has been changed with respect to previous editions (MULTI Bicycle challenge in 2017 and 2018). This paper presents one solution in the context of process management, where universal properties of process types along with task, artifact and actor types together with possible particular occurrences for scoped domains are modelled. We discuss our solution highlighting both the strengths and limitations of our approach, using the MultEcore tool.*

Keywords. Keyword1 • Keyword2 • Keyword3

1 Introduction

Research in Multilevel Modelling (MLM) is continuously increasing. The MULTI challenge was created to enhance discussion and facilitate the contributions within the MLM community. Encouraging researchers to submit solutions to a common challenge makes it possible to compare them and fosters improvements in the same direction. We use the MultEcore tool (Macías et al. 2016) to apply various MLM features which are key to be able to fulfil the criteria established for the MULTI Process challenge (João Paulo A. Almeida et al. 2019). MultEcore enables multilevel 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 in Department (n.d.[a]) and the solution to this challenge can be downloaded in Department (n.d.[b]).

With MultEcore, modellers can configure flexible multilevel hierarchies that can be composed to include new features. This is mainly done by the definition of *application* and *supplementary* dimensions. First, *application hierarchy* can be understood as the main multilevel hierarchy (*base language module* in the context of *language product line* Méndez-Acuña et al. 2016). Second, *supplementary hierarchies* are used to add new dimensions to the application one. Application hierarchies can include several supplementary hierarchies which can also be removed consistently without introducing inconsistencies.

We also take advantage of some of the last features we are working in with regard to the specification of constraints (static semantics) and behaviour description (dynamic semantics) by using Multilevel Coupled Model Transformations (MCMTs) (Macías et al. 2019; Rodríguez et al. 2019a).

* 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: If your submission is based on a prior publication and revises / extends this work, enter a corresponding note here (This work is based on ...) but DO NOT cite the prior work during the reviewing process. INSTEAD provide full citations of all prior publications to the editors during the submission process (use the text field in the online submission system).

The key aspects that characterise our conceptual framework and have been applied to solve the challenge are summarised as follows:

- The definition of multilevel 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* (i.e., to restrict the level at which an element can be used to type other elements).
- Being able to define several supplementary hierarchies helped us accomplish some of the requirements of the challenge. This will be further discussed along the paper.
- Specifying MCMTs allows us not only to define generic rules that apply to the general language defined at the most abstract level, but also to create tailored rules that might apply to one of the domains. MCMTs can also make us of concepts defined in different hierarchies.

In this edition, the challenge concerns the domain of process management, a domain in which one is not only interested in particular occurrences (i.e., "processes" = "processes instances", "tasks" = "task occurrences"), but also in universal aspects of classes of occurrences ("process definitions", "task types") and relations to actor types and artifact types. Note that we stick to the British style, however, we use *artifact* instead of *artefact* to be aligned with the challenge description. Respondents are required to define first, universal concepts for process management, and second, an application of such a conceptualization in the scope of a particular software engineering process. Optionally, one can also capture a different scope for the insurance domain. In order to highlight the flexibility of our framework we provide a multilevel hierarchy where both domains are included.

The rest of the paper is organised as follows. We discuss in Sect. 3 relevant aspects of the case presented in the challenge and in the context of the solution approach. Sect. 4 describes the complete scenario that contains the multilevel architecture built for this challenge. In Sect. 6 we discuss the solution and cover the key points that are required to be explicitly reasoned. We also examine the limitations and the requirements that could not be satisfied. Finally we summarise and conclude the paper in Sect. 8.

2 Technology

Our solution has entirely been modelled using MultEcore (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 (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 and the semantics (behaviour and constraints) is provided by our multilevel transformation language Multilevel Coupled Model Transformations (MCMTs) (Macías 2019; Macías et al. 2019). Using the MultEcore tool, modellers can (i) define MLM models using the model graphical editor; (ii) define MCMTs using its rule textual editor; and (iii) execute and analyse specific models. The execution of MultEcore models rely on a bidirectional transformation of the models into Maude (Clavel et al. 2007) specifications. When we design a multilevel DSML, we first define its syntax/structure with multilevel modelling hierarchies. Then the semantics is specified via the MCMTs.

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. Models are distributed in *multilevel modelling hierarchies*. A multilevel modelling hierarchy in our context is a tree-shaped hierarchy of models with a single root typically depicted 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 level is grouping them by how they define a (potentially) independent modular artefact that is part of the DSML under construction and how reusable and useful these elements can be in that particular level. 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*. 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 multilevel 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. 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 on this). For instance, we might have different hierarchies (physically separated, e.g., different projects in the MultEcore tool) that we want to compose. 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 *Timestamp Hierarchy* is a supplementary one (see Figure 1 and Section 4).

2.3 Instance Characterisation

MultEcore allows for deep characterisation (Atkinson and Kühne 2001) which means that elements can be instantiated not only in its immediate model below, but also further 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. While there exists different potency definitions within the MLM community, we use a three valued By using potencies on elements, we can define the degree of flexibility/restrictiveness we want to allow on the elements of our multilevel 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.

How to model with MultEcore, representations, three-valued potency (min-max-depth), supplementary, etc. Clarification in the document: "Avoid vague language such as "higher level concepts are more abstract" if the inter-level relationship is more specific. If the inter-level relationship is deliberately allowed to be vague, state this explicitly."

How to MT with MultEcore, representations. Also used for constraints?

Naming convention: Verb associations go in third person, we do not use "type" explicitly in the names.

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 we have organised the concepts it presents: in a hierarchy with a top model for the generic concepts related to processes (Fig. **??**, top), from which two branches span. The first branch (Fig. **??**, right) refines these concepts for the (sub)domain of software engineering processes. The second branch ((Fig. **??**, 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 requested. The description of the challenge also requests that 'submitted solutions should include bottom-level instances, at least for key types, exemplifying all attributes mentioned in the challenge description'. So each branch includes a bottommost instance model which illustrates the instantiation of the classes and attributes to define a specific state of a process. To sum up, our contribution is a five-level, two-branch hierarchy. Levels numbered from 0 (on top) to 4 (in the bottom). Recall that L0 excluded from presentation in figure. This fixed, top-most metamodel, together with the fact that we do not limit the number of levels that a hierarchy can have, makes the incrementally downwards numbering of levels a more sensible choice. It is important that we make this remark since many other approaches to MLM, and MDSE in general, use a numbering system that increments upwards, with level 0 at the bottom. Each level in our solution accounts for a different degree of abstraction in the challenge description. In level 1, the generic language for process specification is defined, according to the most abstract concepts from the domain. Level 2 scopes some of the generic concepts defined in the level above for specific domains. At this point, the hierarchy branches into two, for the two domains presented in the challenge, namely software engineering and insurance companies. Level 3 refines these concepts to adapt them to the specific processes within the *Acme* software development company or the *XSure* insurance corporation. Finally, in level 4, specific scenarios of processes for these two domains are created in the corresponding branches, yielding a total of 8 models in our application hierarchy, with the 7 relevant ones depicted and explained in the following. These four levels (ignoring Ecore on top) are closely aligned with the ones proposed by de Lara et al. in the original process case study (see J. D. Lara and Guerra 2018, Figure 4).

Assumption: "X type" is redundant, since they are types already, so we dropped the "type" in our naming.

Assumption: actors and what they can perform is an N:M relationship. That is, one actor class being connected via "perform" would force too much redundancy (each actor allowed to do it should have it): all NxM permutations of allowed people to do something. Therefore, we created a distinction between actor and role, and created the actor - role - task triangle, which addresses P5, P9 and P14 (see next section). This also allows us to easily apply a composite pattern (**todo**) for combined roles, which are suggested in P?. These elements do not affect the general semantics of the models or the alignment with the requirements of the challenge.

In order to support the requirement for time stamps in a minimally invasive manner, we created a hierarchy, which is a perfect fit to such kind of 'aspect orientation' techniques for MLM.

4 Model presentation

Fig. 1 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

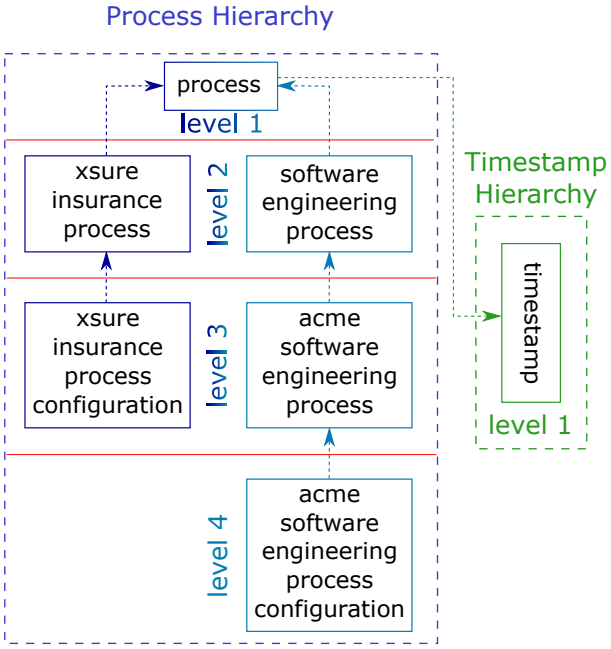

Figure 1: High level System overview

Process Hierarchy. We describe each level in a subsection and start from level 1 (process).

We also describe the supplementary hierarchy (dashed box to the right) Timestamp Hierarchy and the utility and convenience it provides to overcome 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 the usage of this in Section **TODO**.

We only display the cardinalities on references in those cases where it is not the default one (0..*).

4.1 Level 1 - Process

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

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 level metamodel, is indicated in an (light blue) ellipse at its top left side, e.g., EClass is the type of Process. Note the second (green) ellipse at the right of Process that provides it with a supplementary TimeStamp type. Even though we further discuss this in Section **??**, it is worth mentioning that this allows to handle the requirement P19 and allows us to instantiate the lastUpdated attribute on any node of the process hierarchy. It is also important to mention that we only double type the Process node and instantiate the attribute there (lastUpdated=22/03/2021) for the sake of simplicity, and to demonstrate the usability of the supplementary dimension.

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 currently typed by one of the four basic Ecore 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 isCritial as a boolean. These attributes can be instantiated in a lower level with a value, as illustrated in Section **??**. 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 reference, specify their potencies. Potency in attributes is displayed as two numbers as an attribute does not have depth, since first it is declared, and eventually in a level below it is instantiated. The two numbers are specified in front of the attribute name. *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 on elements, we can define the degree of flexibility/restrictiveness we want to allow on the elements of our multilevel hierarchy. The MultEcore's potency implementation combines range and start potencies which are specified via the three numbers clarified above. 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

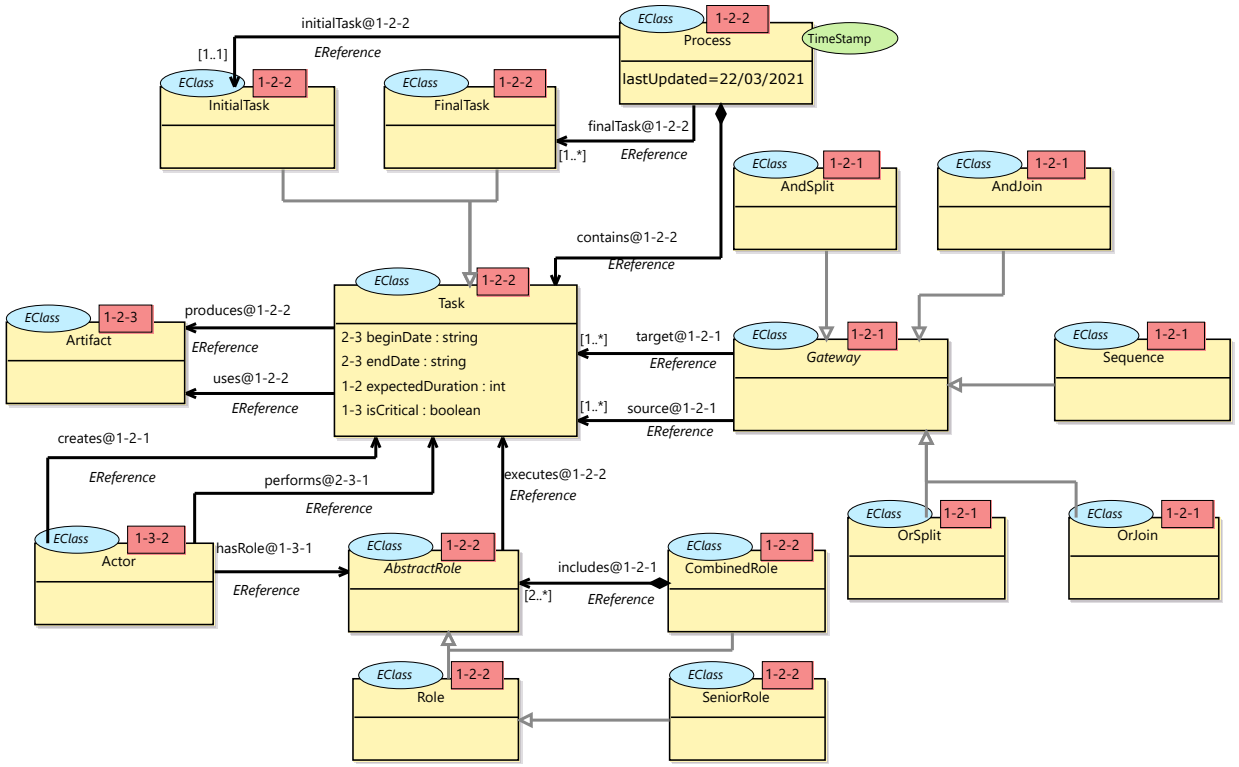

Figure 2: Level 1: Process model

instantiated, or re-instantiated, regardless of the levels where this occurs. For instance, the potency specified for Task is 1-2-2, 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 2 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 ([1..1] cardinality in initialTask reference) Initial Task and can have one or more Final Tasks ([1..*] cardinality in finalTask reference). The MultEcore tool allows us to make use of the *inheritance* relation, being InitialTask and FinalTask subclasses of Task. In MultEcore we can also mark a class, e.g., Gateway, as an abstract class, which cannot be instantiated (note the italics). This means that a Gateway must always be instantiated using one the five subclasses specified, namely, AndSplit, AndJoin, Sequence, OrJoin or OrSplit (right side of Figure 2, and they connect (one or more, depending on the gateway, as indicated in the multiplicity in the source and target relations) tasks between them.

A Task might use and/or produce Artifacts which can be created and performed by Actors. We define *actor types* as Roles. The reference hasRole between Actor and AbstractRole models this. We use for roles the traditional object-oriented Composite pattern (Gamma 1995). We define AbstractRole as an abstract node (italic font in the name). 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 includes reference ensures there are at least two roles combined). Finally, certain roles can perform certain tasks, which is covered

by the executes reference from AbstractRole to Task.

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. 1).

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. 3 and it corresponds to software engineering process in Fig. 1.

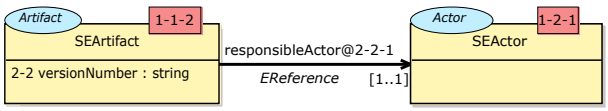

Figure 3: Level 2: Software engineering process model

The creation of this level facilitates the resolution of multiple requirements which are discussed in Section 5. Every software engineering artifact (SEArtifact, which is typed by Artifact in the level right above, defined at level 1, Figure 2) 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 (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 Figure 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. 1. We show in Figure 4 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 João Paulo A. Almeida et al. (2021) (left-hand side of the figure). The complete model that fulfils all the requirements and specifications can be found in Appendix A From this point the described elements are all referred to the left-hand side of Figure 4.

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

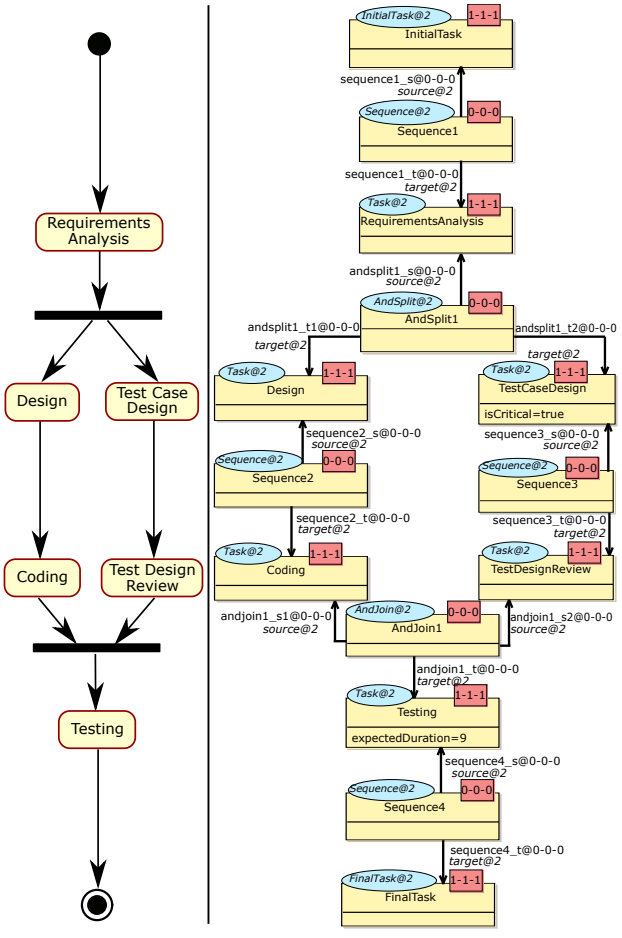

Figure 4: Level 3: Selected parts of the Acme software engineering process model (right-hand side) to compare it with the graphical schema given in the Challenge description João Paulo A. Almeida et al. 2021

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 references, in the types with italic font. One can observe, comparing with the right-hand side representation, that all the information is faithfully represented and easy to track.

Note that some elements' potency, such as Sequence1, AndSplit1, Sequence2, Sequence3, AndJoin1 and Sequence4, is 0-0-0. This is because these elements cannot be further instantiate, which appropriately controls than these elements belong to this level where the general Acme workflow is represented. Also, notice that we instantiate some attributes here (that are further disclosed in Section 5 where the requirements are explained one by one), for example, isCritial wich is set as true in TestCaseDesign node, and expectedDuration=9 in Testing node.

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 to the Acme software development process. In this branch (software engineering domain) this model represents an instance model which contains concrete elements representing a model state of a potential executable scenario. This is shown in the model depicted in Fig. 5 and it corresponds to the acme software engineering process configuration element in Fig. 1. The nodes and relations displayed in this model has been obtained by the information provided along the software engineering domain requirements (**S1, S2, etc.**) from the challenge description document (João Paulo A. Almeida et al. 2021). Notice that all nodes and relations at this level have as potency 0-0-0, since this represents the bottommost model and cannot be further instantiated. We discuss the elements of this model from left to right.

JohnDoe (typed by SEActor@2 is responsible of (cobol_responsibleactor and cobolcode_responsibleactor references) the concrete artifacts COBOL (typed by ProgrammingLanguage, and that has the version number attribute instantiated as versionNumber=1.3) and COBOLCode. Also, COBOLCode is written (the type of the reference cobolcode_written) in COBOL. Besides, the concrete CodinCOBOL (typed by Coding) task uses COBOL as and produces COBOLCode (these two relations are captured by codingcobol_codinguses and codingcobol_codingproduces, respectively.

Ultimately, AnnSmith is an actor (typed by SEActor@2), that has assigned, via the annsmith_hasrole reference, of type hasRole@3, the COBOLDeveloper role. AnnSmith performs CodingCOBOL which the COBOLDeveloper role is allowed to execute (through the coboldeveloper_ . . . relation). Notice that certain types of the elements aforementioned (e.g., Developer or ProgrammingLanguage) are not explicitly shown in Figure 4, as we comment at the beginning of Section 4.2.2, the complete model is shown in Appendix A where all these elements can be found.

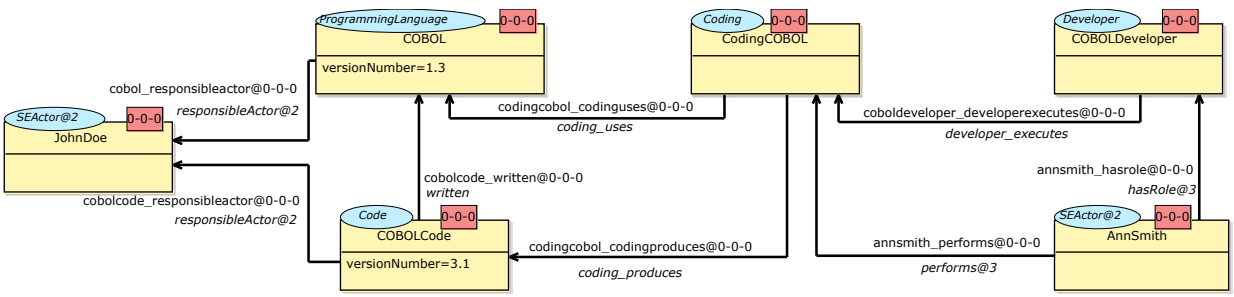

Figure 5: Level 4: Acme software engineering process configuration model

4.3 Insurance process domain

In the challenge description it is described the so-called *insurance domain*. Even though it is suggested to mainly focus on the software engineering domain and the insurance is 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 Section 2.2 of the challenge description document. As one can observe in the left-hand side of the Process Hierarchy in Figure 1, the branch is composed by two models (if we ignore process at level 1) rather than three as in the software engineering domain. Even though we further go into details on this in the remaining of the article, the needs of this domain (again, from the information extracted) do not require the creation of a model that is equivalent to software engineering process model (level 2 in Figure 1). Instead, the level 2 of the insurance branch directly corresponds to the XSure company (xsure insurance process model). This also clearly demonstrate the flexibility of MultEcore where the length of the different domains does not need to be the same.

4.3.1 Level 2 - XSure insurance process

Since for this domain it is not explicitly stated information of any kind that would affect any company within the insurance domain, we do not create model with elements equivalent to those created for the software domain such as SEActor or SEArtifact.

The xsure insurance process model (corresponding to the element to the left of Figure 1 in level 2) represents the workflow of the Claim Handling process. For illustrative purposes we show a fragment of the model in Figure 6 and point the reader to Appendix B for the complete model.

We describe the model from top to bottom. A ClaimHandling (of type) Process is composed by all the tasks depicted in the model. To facilitate readability, we only show the containment relation connected to AssesClaim so-called claimhandling_contains2 since it is used in the level

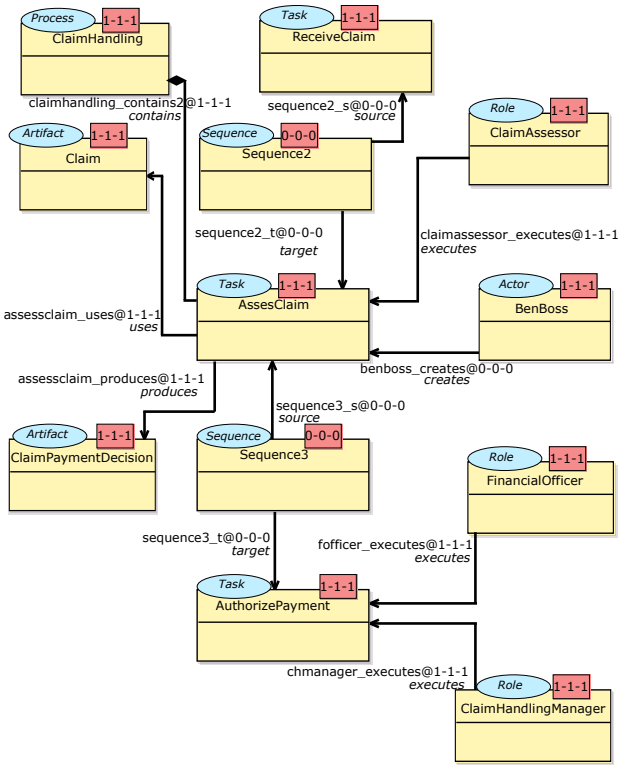

Figure 6: Level 2: XSure insurance process model

below that is described in Section 4.3.2. The reader can find all the remaining containment relationships in Figure 15. A ReceiveClaim preceeds an AssesClaim which are connected via Sequence2. To proceed, an AssesClaim uses a Claim (Artifact, to the left top of the figure) and produces a ClaimPaymentDecision. Also, AssesClaim is created (via benboss_creates relation) by BenBoss, who is an Actor.

Note that a requirement of the challenge forces that any *Task type* (e.g., AssesClaim) must be created by an actor. The full model of the Acme software engineering process in Appendix A captures how BobBrown creates all the task, but to facilitate the readability we do not show those relations neither in Figure 6 nor in the figure in Appendix B.

Furthermore, ClaimAssessor is a Role which is allowed to execute AssesClaim tasks. AssesClaim leads to the following task, AuthorizePayment, connected by the Sequence3 gateway.

Finally, both ClaimHandlingManager and FinancialOfficer roles are allowed to execute AuthorizePayment tasks.

4.3.2 Level 3 - XSure insurance process configuration

As stated before, the xsure insurance process configuration model placed at level 3 (see Figure 1) represents the instance level in this particular branch, i.e., representing a concrete scenario and therefore a non-instantiable model (notice the 0-0-0 potencies). The model is depicted in Figure 7 and the information represented has been extracted from the **PX** requirements described in the challenge description document.

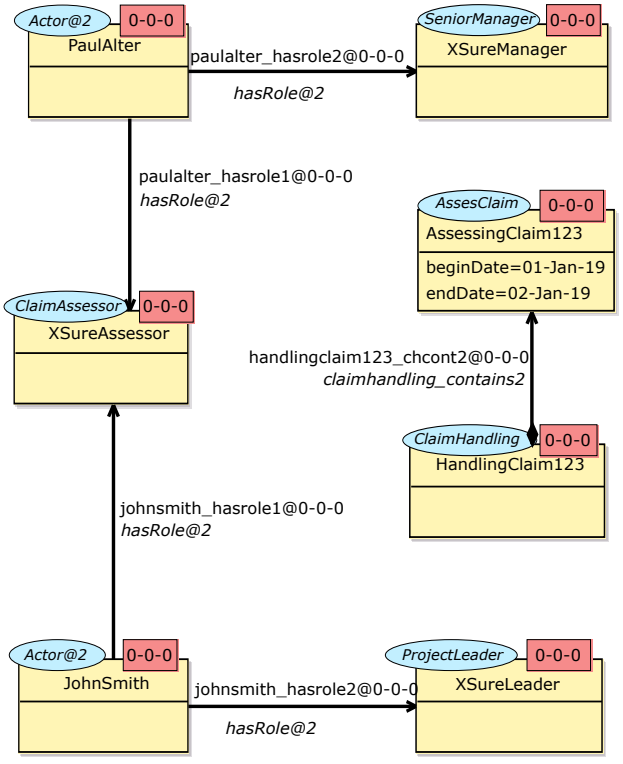

Figure 7: Level 3: XSure insurance process configuration model

An instance of the process ClaimHandling, named HandlingClaim123, is defined at this level and could be interpreted as the concrete implementation of one of the subsidiary office *123* of the XSure company. It 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 have a second role each of them, 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. We denote a multilevel hierarchy as the *main* or *default* one and call it *application hierarchy*, since it represents the main language being designed. In this case, this corresponds to the Process Hierarchy which include both the insurance and the software engineering branches in Figure 1. An application hierarchy can optionally include an arbitrary number of *supplementary hierarchies* which add new aspects to the application one. The supplementary hierarchy feature has been applied in different ways: (i) to complement a main language with additional non-functional features, for instance, data types (Rodríguez et al. 2018) or extra information to augment the information of a node (Rodríguez and Macías 2019) and (ii) to power up instance elements, where composition of application and supplementary hierarchies could be carried on (Rodríguez et al. 2019c).

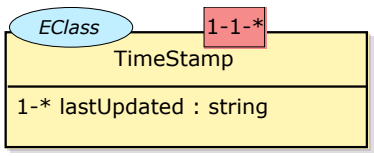

Figure 8: TimeStamp node

Supplementary hierarchies can be added/removed without changing the context or creating inconsistencies in the application one. As hinted in Fig. 1, 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, so-called Timestamp Hierarchy in Figure 1 consists of a single model, called timestamp which has one single node called TimeStamp, as shown in Figure 8. This node has declared the lastUpdated attribute of type string. It is worth to remind 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 be doubled-typed with TiemStamp and furthermore instantiate lastUpdated to give it the desired value (as we showed in Process node at the top of Figure 2) where the attribute is instantiated as lastUpdated=22/03/2021. We have decided to show this functionality just for this element for illustrative purpose, but as said, any node of the models depicted in Figures 2, 3, 4 (and its full version in 14), 5, 6 (and its full version in 15) and 7.

4.4 Cross-level constraints

As introduced in Section 2 MCMTs can be used to specify the dynamic semantics for the definition of behavioural descriptions. In previous work we have proven 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 execution of static semantics, i.e., constraints to check some structural and semantical correctness of the constructed multilevel hierarchy ix 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 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 behaviour we want to acquire in our model and thereby the next state of the system. There is a match when what we specify in the left-hand side (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 is a so-called "Check mode" that behaves differently than conventional MTs. In this mode, the goal is to find a correspondence in the models through a two-steps procedure. Instead of having a model that evolves or change to a new state as it is done for specifying the behaviour (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 multilevel 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. 9 shows a MCMT rule in Check mode to satisfy such a constraint.

The META block allows us to locate types in any level of the hierarchy that can be used in FROM and TO blocks (separated by a black horizontal line). It is worth to mention that the two levels specified (the META and the FROM/TO ones) in this rule are not required to be consecutive and the would match, for instance, if we consider the right-hand branch on Figure 1, level 1 and level 4, 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.

With respect to the application of MCMTs to define behaviour, we do not want to find a match and then change the model (FROM $\rightarrow$ TO),

but to detect if both submodels of the hierarchy models are found (first check that FROM matched, then check that TO matched as well). If so, the constraint is satisfied, otherwise it is not. We see these rules as potential artefacts to be used for performing model repair.

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

On the other hand, we allow the type on the variables to be transitive (i.e., indirect typing). A first correct match of the rule comes when an element, coupled together with its type, fits an instance of Actor (a) that has a relation of type performs (p) to an instance of Task (t). Also there must exist an instance of Role (r) that is linked to t via the e reference of type executes. Note that there are two dashed boxes surrounding certain elements. One must take into account that in different scenarios there could be connected an arbitrary number of tasks that an

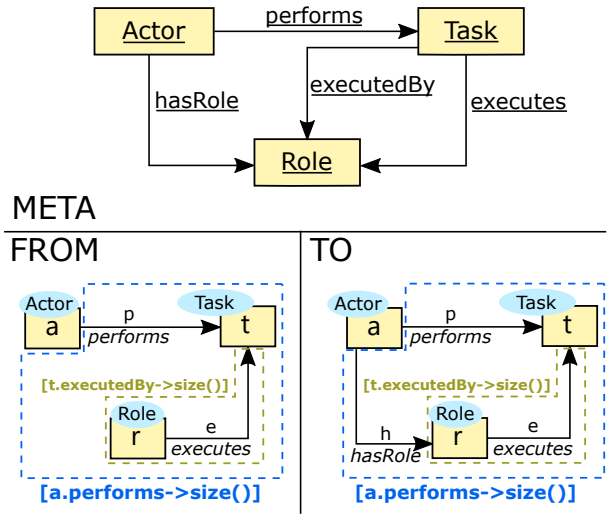

Figure 9: Constraint satisfying requirement ***P17***

actor can perform, and that several roles can also be allowed to execute a certain task. To cover all the permutations with a single rule we use our boxing mechanism that allows us to replicate elements at runtime on demand. Boxes may appear in both sides of the rules, boxes may 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 the computation of the cardinality of the box, i.e., the number of times it has to be replicated, for the manipulation of attribute values (not used in this rule) and for the specification of conditions (not used in this rule), which greatly improves the expressiveness of the tool. 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 allows transitive typing, which allows for the matching candidate to match any type that it is transitively (in intermediate levels) typed by performs or ultimately performs. In practical terms, the [a.performs->size()] means that the size of the collection is given by the number of tasks connected to the matched actor (a) via references of type performs.

Once all the boxes have been unfolded for the FROM of the rule, and there has been a match of all the (unfolded) variables, this partial matching is saved, and their are used in the TO block, to see whether what it is specified in it is also found. In this case, we are trying to find for each task that an actor is performing an existing link (h of type hasRole, at the bottom right of Figure 9) between the actor and some of the roles that are allowed to execute the task.

4.5 Operational semantics

We have demonstrated how MCMTs could allow to handle cross-level constraints to check the structural correctness of the multilevel hierarchy

(Section 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 its first version theoretically presented 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 the model to have an actual execution mechanism that permits to evolve the instance models. To do so, we rely on the Maude System (Clavel et al. 2007; Durán et al. 2020) that is a specification language based on rewriting logic (Meseguer 1992), a logic of change that can naturally deal with states and non-deterministic concurrent computations. A preliminary version of the infrastructure we have implemented (and that still is being improved) was presented in Rodríguez et al. (2019b). In such a version, the multilevel hierarchy and the set of MCMT rules was transformed into a functional Maude representation that could be executed using the Maude console environment. The results had to be brought manually back one by one, 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 **todo**. The current state of the infrastructure that connects MultEcore with Maude relies on a bidirectional transformation that takes MultEcore textual specifications and automatically generates Maude specifications, and then takes the XML output files that Maude produces as result of performing execution, and automatically translates them into MultEcore models that graphically and automatically displayed. The Maude part is handled by a background process which completely hides the Maude part making it aware for the modeller. The current state of the MCMT rules allows us to specify nested parametric boxes to handle submodel collections both in the left- and right-hand sides. Furthermore we can perform attribute manipulations, specify additional conditions and define the boxes cardinalities with a basic support for OCL that has been added. This support is still basic and limited, and we aim to provide a full support for OCL in future work.

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. It is worth mentioning that we do not aim to show a realistic behavioural simulation where all the elements are involved trying to recreate a real enterprise scenario but to demonstrate the (vertical and horizontal) flexibility and applicability of the MCMTs (how the same rules can be applied to both domains). Even though the challenge description do not describe any behavioural or executional aspect of the case study, we believe it is an important part of the modelling process. Therefore, as an illustrative example and to open this line for future Multilevel Modelling challenges, we have sketched five simple MCMT rules that involve the creation of new tasks at the bottommost levels (level 3 for insurance and level 4 for software engineering) through the information of corresponding gateways connected to the Task types at levels above. The set of rules help to handle different cases regarding the different gateways or the initial and final special tasks. In the following, we describe one of the rules, but the remaning ones can be depicted in

We shown in Figure 10(a) a MCMT rule to create several output tasks from an input arc where their types are connected via an *And join* gateway. Similarly as in the rule shown in Figure 9 in Section 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

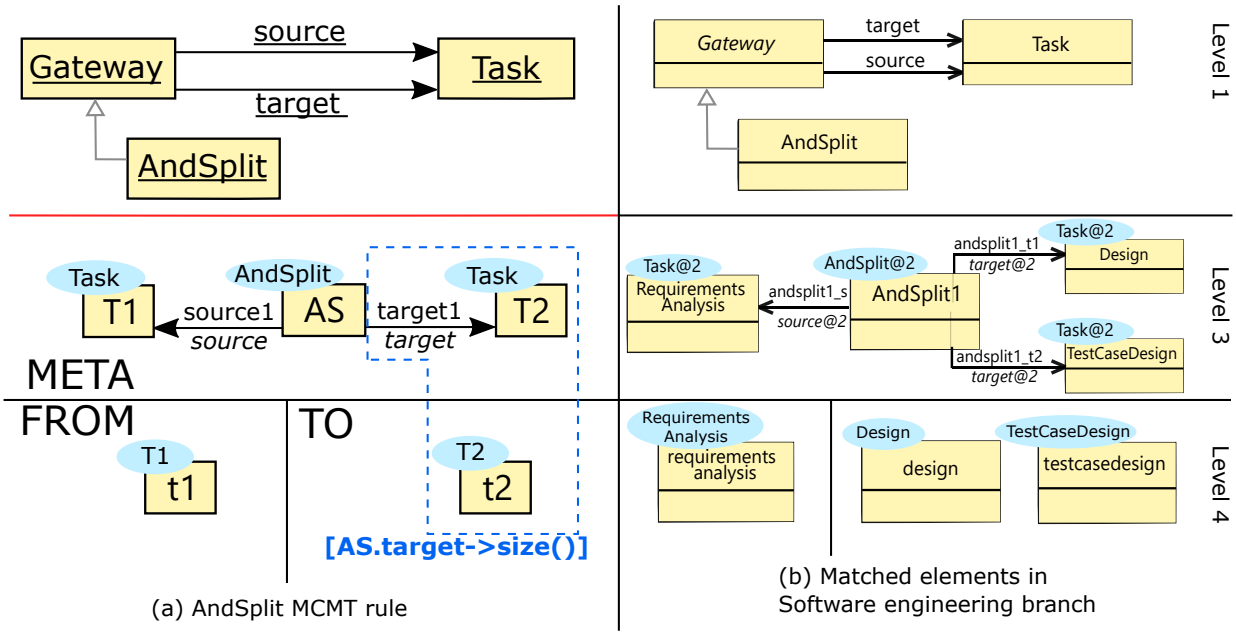

Figure 10: (a)MCMT rule to create new tasks based in their types connected via And join gateway. (b) Matched elements in the Software engineering branch

for the insurance branch. Finally, in the FROM block, we identify a single `t1` task which type (`T1`) would be the input of the corresponding And split. Then in the TO block we remove the matched `t1` and create the new `t2` tasks which types are the outputs of the and split. Note that each particular company (e.g., Acme) can establish an arbitrary number of output tasks from the and split. To make a generic rule that works for any number of output tasks, we encapsulate `target1` and `T2` from the META block and `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 gateways 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 types of each produced `t2` come correctly one by one with the information allocated in the level above.

Figure 10(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.

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 the with a single element named `initialTask`. The five MCMT rules specified allow us to reach an `finalTask` instance following each of the workflows defined

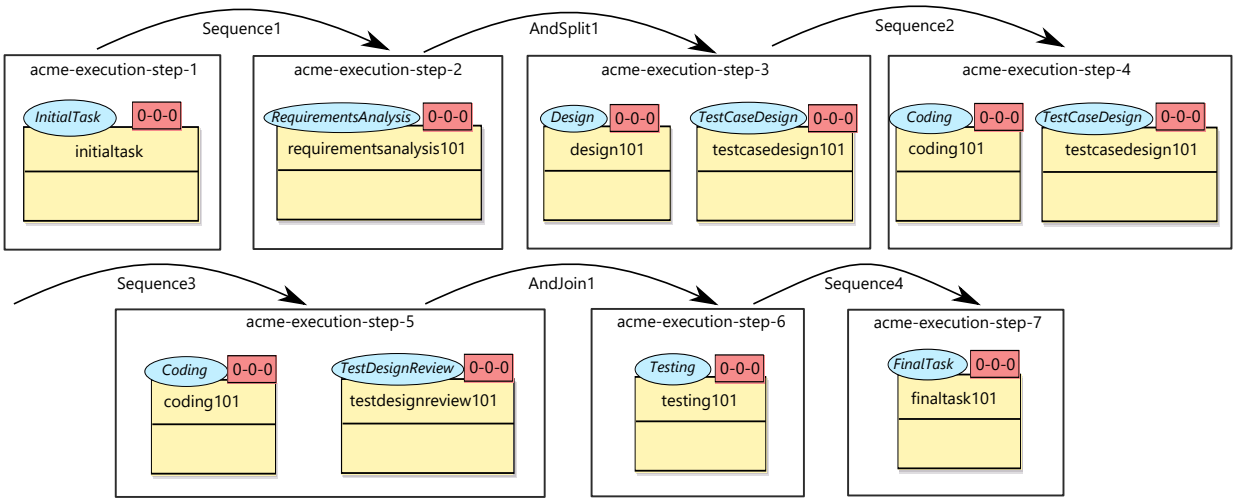

Figure 11: Acme software engineering obtained states by applying subsequent MCMT rules

in Figures 14 and 15. Note that the executions are only concerned about Tasks and Gateways, and do not consider other elements such as Artefacts or Actors.

Figure 11 shows the seven execution model 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 so-called acme-execution-step-1 with the initialTask node. To its right, obtained by applying the rule that triggers the Sequence1 gateway we have the acm-execution-step2 with the requirmenetsanalysis101 node. Note that this number appended to the name (and that its present in the rest of the states), is generated using a *Counter* object that is given the Maude representation, which value attribute gets increased every time a new identifier is created. 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 (these elements, e.g., Sequence1, AndSplit1, etc.) can be found in Appendix A.

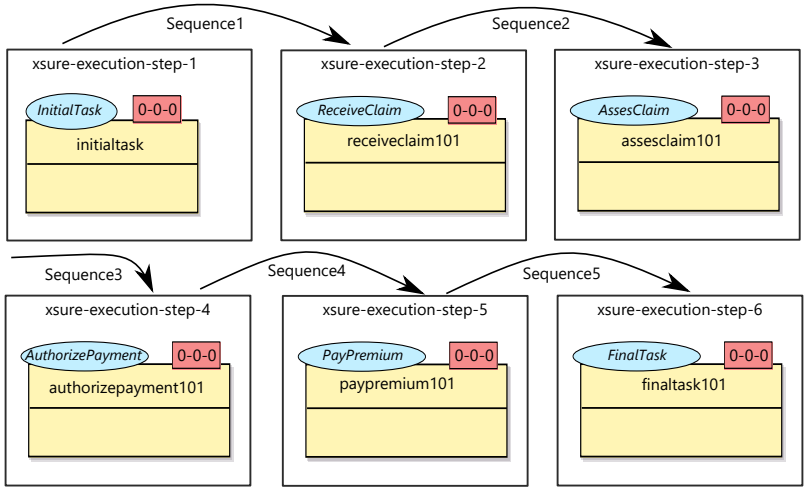

Figure 12: Xsure insurance Claim Handling process model states obtained by applying subsequent MCMT rules

One can observe at the bottom right of Figure 11 how we finally reach an instance finaltask101 representing the end of the workflow.

Likewise, and demonstrating the horizontal flexibility of the MCMTs, Figure 12 represents six model states produced by the execution engine by applying the same set of rules that we had for the software engineering case. Now, the six models represent different states within the insurance domain. Similarly, the workflow established in the corresponding level above (the XSure insurance Claim Handling process) is defined in Appendix B. The process is quite similar as for software engineering, there exists an initial task (top left of Figure 12) 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.

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 (João Paulo A. Almeida et al. 2021, Section 2.2.). We preserve their original name format (PX, with X a number) for easy traceability.

P1
This is done by the definition of Process and Task, and the containment relationship from the former to the latter. 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. In the figures we might not show all the details specified in the requirements, e.g., hide the composition relation of each task with the corresponding process.

P2
This requirements specifies gateways, which can be of different kinds. It is a fixed set, and semantically we consider that they belong to the same level than, task, process, etc. Moreover, they should stay in Process since they are common to all processes. Hence, we choose inheritance here, since MultEcore can do both. Gateway is abstract, the rest are not. If we wanted to add new ones, we can still do it, though.

P3
We also do this via inheritance, for similar reasons as in the previous requirement. Although we could reuse the contains relation, we define specialised ones to easily navigate to the initial and final tasks. More importantly, these relations define different cardinalities to enforce a unique initial and at least one final per process. No inheritance among relations because it is not supported in MultEcore.

P4
We define Actor and the creates relation, which is different from performs, discussed in the next requirement. The example instances are also used in the insurance branch.

P5
We add another relation performs from Actor to Task. However, this is not enough to model which types of actor can execute which types of tasks. As discussed before, we split the concept of actor as an actual person (Ben Boss) and as a specific role that a person may play (claim handling manager). For the purpose of fulfilling this requirement, the way we model 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 Role node plus the hasRole and executes relations, and the semantics are encoded in that "triangle".

P6
We could solve this by creating yet another relation assigned between actor and task. But since we define roles to fulfil other requirements, we can simply take advantage of the triangle and create a role that is assigned to both actors, and cover this requirement without defining new elements. This is also more flexible and makes more sense semantically (there should be some common ability/permission/status/whatever that makes those people suitable to perform the task). Again, examples used in instances of insurance branch, and we add plausible roles to the actors to complete the model.

P7
We define Artifact and the relations uses and produces, and use the examples in the insurance domain branch.

P8
We include the expectedDuration attribute in Task.

P9
Thanks to the triangle we can avoid creating a subclass for Critical Tasks. The attribute criticalTask that we add to Task serves us to define a MCMT rule to constrain this. See **??**. Since the information about what an actor can do is not stored in Actor itself but in Role, we create the subclass SeniorRole of Role. 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 (indicated by the executes relation).

P10
This is trivial since we allow for multiple instantiations.

P11
This is already covered in P1 by the contains relation from Process to Task, and it is also covered in the instances by the instantiation of these three elements.

P12
We include the corresponding attributes in Task.

P13
We include the uses and produces relations from Task to Artifact, and the performs relation that we mention in earlier requirements.

P14
This is trivial by simply instantiating Artifact, Task and the uses and produces relations.

P15
Thanks to our separation of actors and their roles, this is also trivial. And we also avoid multiple typing, which is at best tricky and simply undesirable in many cases, including MultEcore, where it is not allowed (within the same hierarchy, i.e. in the same domain, refer to supplementary explanation earlier).

P16
Same as with Role, we could use a composite. We choose not to show it in the model to keep it cleaner, but node Artifact could be replaced by AbstractArtifact, together with the rest of the elements shown in Figure **??**. In such a way, instead of using multiple typing (which, as argued before, is not a desirable nor possible alternative in MultEcore), we could combine SimpleArtifacts into CombinedArtifacts, and use those combined artifacts as a replacement of multiple typings.

P17
Again, the "triangle" allows us to fulfil this requirement without adding any new elements. An actor which performs a task must also have the role that is allowed to execute that task. 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.

P18
The nature of inheritance in MultEcore allows us to model this. Since we split 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). 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 reference to a task from the specialised role (parent), and any actor having the specialising role would be allowed to perform that task. However, since the description mentions the possibility of "senior" versions of the different roles (also 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: 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. The constraint that enforces that only actors with the right role

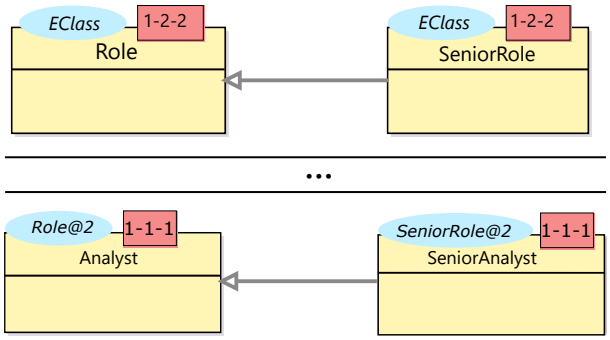

Figure 13: Fragment of the process multilevel hierarchy showing the P18 requirement fulfilment

might execute a particular task also ensures that this requirement is fulfilled (see Figure **??**).

P19

The gist here is defining the attribute somewhere so that it can be instantiated no matter how the hierarchy grows (in depth, width or number of elements in the model) without forcing the modeller to add more definitions of the attribute, or typing/inheritance relations, or such things. We have considered four possibilities here. First, adding it to every node without a parent node in process model. But that would forbid the actual elements in that model from instantiating it, so it is not a valid option. Second, adding one level on top of process for the definition of a node which contains the attribute definition, and type everything in process by it. This does not make sense semantically, is an ad-hoc solution and we have better options. The last two options are based on the use of our supplementary typing mechanism to separate concerns/aspects, 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 away, we can define a supplementary hierarchy with a single model (two, counting Ecore on level 0), which contains the Node ?? with the timeStamp attribute with 1-1 potency and 1-1 multiplicity (the description says that they *must* have it). So, the third option we consider is adding this Node as supplementary to every single one in the application (main) hierarchy. However, this is not ideal, since every single node we add to the hierarchy needs to be double-typed with this supplementary type to be able to instantiate the attribute. Finally, the fourth option which we actually implement is an improvement of the previous one: we do the same, but give the attribute potency 1-* and only add ?? as supplementary to Process, Task, Gateway, AbstractRole, Actor and Artifact. We only define the attribute once, we "link" it six times (supplementary typing) and it is already available everywhere thanks to inheritance (in process model) and potency (in the rest of the hierarchy). Moreover, it would still be available in new branches, new models in the existing branches and every element that we define using the types of Process. If would only be needed to add manually if we instantiate EClass, which most likely could happen in model process (which should not change much anyway). So we believe we present a nearly-optimal solution for this requirement.

Secondly in this section, we discuss the requirements related to the requirements which are specific for software engineering processes (Section 2.3. in the description). We reproduce in MultEcore the diagram shown in Figure 1 in the description, as we show in a side-by-side comparison in Figure **??**. Note that the instances of Sequence are depicted as nodes and two arrows to indicate their source and target. Using this figure as a starting point, we add the different nodes and edges that we require to fulfil the requirements. Again, we refer to them with their original names of the form SX.

S1

Requirements analysis already exists in the description's figure, so we added Analyst:Role and RequirementsSpecification:SEArtifact (not Artifact due to S10, which also apply to some of the following requirements), and the corresponding relations according to the process model (Figure **??**).

S2

We include SeniorAnalyst:SeniorRole, and the corresponding instance of the executes relation. The role does not need to be connected to an actor

necessarily in this model, although we could do so. But we choose not to overload the models with additional details to simplify their description and visualisation in this paper. We also include TestCase:SEArtifact and instantiate the produces relation to indicate that it is a product of test case design.

S3
To fulfil this requirement we include the nodes Developer:Role, Code and ProgrammingLanguage, the latter two being instances of SEArtifact. We connect these nodes to the coding task by instantiating, respectively, the relations executes, produces and uses.

S4
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 pertains two software-specific artifacts, it does not have a type in the process model. For such scenarios, MultEcore always allow to create direct instances of an EClass or EReference. In this case we use the latter as the type of written. Although this construction differs conceptually from linguistic extensions **todo**, its practical usage is quite similar to those **todomaybe**.

S5 and S6
We group together these two requirements since they are pretty similar. Coding in COBOL, as an instance of the coding task, belongs naturally in a level below where the Acme software engineering process, since the latter deals with coding as a generic concept, as the original figure in the challenge description shows.n Therefore, the node CodingCOBOL:Coding is declared in the bottom-most model of the software branch of our hierarchy: acme software engineering process configuration (acme configuration for short from now on). The same reasoning can be applied to COBOLCode:Code and COBOL:ProgrammingLanguage. To complete the model, the relevant instances of the coding_uses, coding_produces and written are used to connect those three nodes to each other, modelling the semantics of both requirements.

S7
The fulfilment of this requirement implies creating an instance of actor in the configuration model. Due to the refinement of Actor (from process model) into SEActor (in software engineering process model) 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 (that relates actors to tasks) to indicate that she carries out the task coding in COBOL. As we 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 reference. 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 mentioned in **??**.

S8
Testing is already present, so we add the nodes Tester:Role@2 and TestReport:SEArtifact to the model acme process. We instantiate the references executes and produces (from two levels above) in order to connect them, respectively, to Testing.

S9
At first glance, this requirement could be satisfied in the model software process (level 2). However, we believe that it corresponds to acme process, since it only concerns Testing, which is declared on that level and may or may not be defined or used in the same way in the potential software processes defined in other hypothetical software companies. This way, we also avoid the need for a constraint that would check that Testing is associated with (only) some instances of SEArtifact.

Therefore, we include in model acme process a node TestReport:SEArtifact that is connected to the existing Testing and Code via new reference called isTested and testing_produces:produces@2. 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 not obstacle if one wanted to create more relations like it from other instances of SEArtifact to other test reports (or even the same, if one wanted to model a test report that contains info about several tested artefacts).

S10

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. Here, 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 define the required attribute versionNumber of type string, which should be instantiated in the bottom-most level of the branch, i.e. model acme configuration in level 4. Hence the potency 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 SEArtifacts 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

We interpret that 'this software development process' refers to a specific instance of a software process. That is, the model in Figure 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 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. While this construction for actors might seem undesirable at first, we argue that it removes the need for cross-level relations and that it does not require 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 this 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
Testing in model acme process instantiates the expectedDuration integer attribute to 9.

S13
We instantiate the isCritical boolean attribute to true in TestCaseDesign in model acme process. We also create node SeniorAnalyst:SeniorRole and connect it to that 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 a test design review validates the test case design is already represented in the original workflow as a sequence of the two tasks.

6 Assessment of the Modeling Solution

6.1 Basic modelling constructs

Nodes and relations are the basic building blocks, and they are contained in models. Models try to be as independent from each other as possible, only related by typing relations. This favours to easily add and remove intermediate models, e.g. software process could be removed and the types in the models below just move to the type of those types, e.g. SEActor to Actor and responsibleActor to EReference. Potency plays a big role, as discussed later. Combining inheritance with typing also allows us to choose whichever construction is more flexible, understandable and aligned with the requirements, e.g., the composite of roles.

6.2 Levels

Levels as organisational tool, and relation is nodes-to-nodes and relations-to-relations. It has the meaning of "my type defines my structure", meaning which relations can be defined and to which other nodes, which attributes can be instantiated, which elements can inherit from, etc. Due to potency, these can jump over levels. Not necessarily adhere to classification with all its implications, since we prioritise flexibility and conciseness, but quite similar. [Copy and adapt the segregation and cohesion discussion from thesis or other papers].

6.3 Number of levels

As stated before, hierarchies in MultEcore are unbounded, so it could grow down 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. But this would pollute that model, which is supposed to be generic and unaffected by the particularities of any sub-domain. We also 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 needed later. So we kept as flexible as possible, and tried to use levels for what are, in our understanding, clearly-defined partitions of the domain (for the software branch): processes in general, software process, the software process of a particular company, and the state of such process at a specific point in time.

6.4 Cross-level relationships

Cross-level relations break modularity, and would disfavour some benefits of our approach, e.g. flexibility and reusability. Also our current formalisation **todo** does not allow them. We choose not to use them and have not found any case in which they are better than an alternative construction that does not use them.

6.5 Cross-level constraints

The expressive power of MCMTs allows us to use them to define any type of semantics. In other words, not only can we specify dynamic semantics to describe the behavioural aspect of the modelled system but also define static semantics that check the structural correctness of the multilevel hierarchy. As discussed in Section 4.4, 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 specification of the right-hand side. However, 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 multilevel sub-hierarchy patterns that have to be found for the constraint to be satisfied.

6.6 Integrity mechanisms

Two parts: integrity and repairing. Integrity are identifying that something is broken. We have both formal constructions and tool checks to prevent: cyclic inheritance, cyclic typing, potency-violating typing, invalid inheritance, multiplicity violations (relations and attributes), ... Repairing we have less because is more of a tool-related thing. We have: fixing potency to 0-0-0 if any is zero or depth to one less if it is higher, ... We could add: fixing types if they disappear (jump up or to EClass/EReference default), ...

6.7 Deep characterisation

We make heavy use of potency, which in MultEcore is three-valued to allow for even richer deep characterisation. Default is 1-1-*, but here we want to control depth, while allowing to instantiate some elements lower than 1 or even constrain them to be instantiated in the level below.

6.8 Generality

We believe so. Almost no redundancy, process is fully applicable to other domains (we illustrated with the optional insurance domain). Software process could be used with other software-related companies. The ACME process could be instantiated for other points in time of the same enactment (refer to MCMTs?) or enacted differently for other departments of the same company that adhere to the same process.

6.9 Extensibility

Indicate whether there are formalisms to establish the semantics of the MLM technique and/or tools that support the presented solution. There are: GT + CT and MultEcore

Discuss model verification (e.g., consistency analyses) or other quality assessment mechanisms supported by the MLM technique employed. We can rely on MultEcore's internal checks (e.g. no duplicated, no wrong typing, etc.) and EMF's verifyer (e.g. multiplicities)

Our solution is able to properly separate the different levels of abstraction of the domains included in the challenge, keeping the common concepts at level 1 and then branching their refinements for two different, more specific domains, in levels 2, 3 and 4. This way of organising concepts from a domain has been used several times already in MultEcore Macías 2019, including the first edition of the bicycle challenge Macías et al. 2017.

Any of the models at levels 1, 2 and 3 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 models at level 4 represent a specific state of the process, e.g. *John Smith, who is a Senior Manager, is reviewing claim 123* for the insurance domain; or *Ann Smith, who is a Developer, is using COBOL version 1 to implement the third version of a particular piece of 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 asses whether the specified process, task distribution, workload, etc. are likely to succeed or will probably lead to time and budged overruns.

Deep characterization is a key feature when defining multilevel structures. The more abstractions levels we have, the more necessary becomes to allow, on one hand, be precise enough to prevent undesired behaviour and, on the other hand, to be flexible enough, specially when defining a family of languages that might have several domain-specific variants within the same hierarchy. we fully rely on our definition of *potency* consisting of three values to tackle deep characterization. One can see that most of the nodes and references in level 1 (in Fig. **??**) have as potency `1-2-*`. This constrains the possibility of instantiating general tasks in the level 4, which always belong to a specific domain. With such a potency, a task must be instantiated either in level 2 or level 3.

Another interesting example is shown in level 2. For instance, in the software engineering process

(Fig. 3). While SEArtifact might only be instantiated in the level right below (level 3), the fact that a software engineering actor (SEActor with potency 2-*-*) is responsible (responsibleActor with potency 2-*-*) for a specific artifact belongs, at least, to the fourth level of abstraction. Furthermore, it does not make sense that a software engineering artifact type (which is defined at level 3) has a concrete version number. Thus, versionNumber attribute potency is restricting that it can only be instantiated two levels below (2-2 potency). Note that we relax SEActor and responsibleActor potencies, but not in versionNumber. In case of a fifth level was further introduced, one could instantiate the former two but not the latter. This configuration does not affect the current distribution of the hierarchy and we show both possibilities (relaxed vs restricted potency) with a merely illustrative purpose.

Regarding integrity mechanisms, we do not yet include co-evolution mechanisms in MultEcore. This limitation is purely instrumental, but it implies that we cannot yet deal with the co-evolution of models and their instances. If the user changes or deletes an element in the top models, the instances of such types will become invalid and they will eventually be deleted by the tool, unless their types are updated. MultEcore does feature basic syntactic checks to ensure that the types, potency values and relations among nodes are valid. The formalisation behind our framework, based on Graph Theory and Category Theory, can be found in Wolter et al. 2019 and Macías 2019.

The last interesting matter of discussion is the instantiation of the Actor element. Due to requirements **P4**, **P6**, **P13**, specific actors needed to be instantiated in two different levels. In level 3, in order to specify who created a particular task, due to requirement **P4**. And in level 4, so that we can related a task to the specific actor who is performing it, due to requirements **P6** and **P13**. Our solution for that, due to the lack of cross-level relations in MultEcore, is allowing for the instantiation of actors in both levels. This may cause that, in some cases, an actor who creates tasks but also executes tasks (the same or different ones) appears duplicated in two models in two different levels of the same hierarchy. However, we believe that the negative side effects of this limitation are not that critical since it is not likely that a person is in charge of organisational matters and, at the same time, of performing the job. Moreover, interpreting the models (for code generation, querying of the models, etc.) without considering the duplicated actor as two different people is trivial, since the name of the element is used as an id. Therefore, we simply need to use the duplicated ids as a method to detect that they represent the same real person.

As for the comparison of our approach with other MLM techniques, we believe that the main aspects that differentiate MultEcore are the following.

- A framework not based in the OCA architecture, which makes the approach independent from a fixed linguistic metamodel. To the best of our knowledge, only FMMLx Frank 2014 follows a similar paradigm to ours, that the authors call "golden braid".
- Three-valued potency that allows for fine-grained control of the instantiation of nodes, relations and attributes. This concept is able to unify consistently the kinds of potency defined, among others, in Atkinson and Gerbig 2016a and J. d. Lara and Guerra 2010.
- The specialisation construction, defined to be compatible with typing and potency in a multilevel hierarchy.
- The concept of supplementary hierarchy to introduce aspects in our models that are not strictly related to the domain that is modelled in the "main" application hierarchy (e.g. language and technological domains).
- MCMTs to exploit the multilevel capabilities of our framework, both for model transformation and constraint specification.

Regarding the questions that the challenge description explicitly asks respondents to address, we include them in the following, together with our responses, as a summary of this section.

'*Does the response address the established domain as described in Sect. 2* [in João Paulo A. Almeida et al. 2019] *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 supplementary hierarchies). All of these concepts are important multilevel features that this submission showcases.

'*Does it evaluate/discuss the proposed modeling solution against the criteria presented in Sect. 3* [in João Paulo A. Almeida et al. 2019]*?*' The main part of this section is dedicated precisely to the discussion of those criteria, in the same order that they are enumerated in João Paulo A. Almeida et al. 2019, so that we can make sure that this question is properly addressed.

'*Does it discuss the merits and limitations of an MLM technique in the context of the challenge?*' The rest of this section above is dedicated precisely to such discussion, and we have addressed both the benefits of our approach and the shortcomings we have found, together with suggestions on how to overcome them.

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 (João Paulo A. 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 to define hierarchies of level objects (called *most-general* (MGI) instances). 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 due to the theoretical formalisation as the only allowed cross-level relationship is the *typing* relation. On the other hand, Jeusfeld argue that certain requirements, such as P17 can no be completely fulfilled as they would have to extend their specification by Telos rules. Our multilevel transformation language, the MCMTs that allows us to specify multilevel constraints. The most-general instances idea replaces the well-known *potency* mechanism present in level-adjuvant approaches. In concrete, our 3-value potency specification allows us to be both generic and precise depending on the particular needs. Such level of precision remains a bit blurry to us regarding the solution in Jeusfeld (2019b), where they also have to make the explicit separation between `Task` and `TaskType` which we see unnecessary in a multilevel hierarchy context.

Somogyi et al. (Somogyi et al. 2019) also contributed with their Process 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 support 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 to the modeller to construct invalid models, fact that is more difficult, for instance, in our approach, where the hierarchy of models is clearly constructed. Furthermore,

the sanity checks that potency gives facilitates the modeller to always be sure that the model under construction is correct. Also, DMLA does not explicitly support, for example, inheritance (even though authors argue it can be simulated, and they had to due to such a lack, that would have brought more natural solutions to certain requirements), feature that MultEcore naturally implements.

The two solutions discussed above were the only ones published along with our MultEcore response in 2019. Even though in 2018 there was another challenge case, namely the Bicycle Challenge (see Committee 2018), we find 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 discuss in the previous paragraph.

Lange and Atkinson's solution (Lange and Atkinson 2018) was constructed using the mature tool Melanee (Atkinson and Gerbig 2016b). Melanee is one of the most OCA-based (Atkinson and Kühne 2005) advanced tools 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 distribute 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 is having relevance in two different abstraction levels. If we take as reference the right-hand branch of the multilevel hierarchy depicted in Figure 1, while an Actor can create Tasks (types) in level 3 of this branch (see, for example, Bob Brown at the right side of Figure 14) it can also perform concrete Tasks such as Coding in COBOL performed by Ann Smith (see bottom right of Figure 5).

Finally, regarding our submission to the Challenge in 2019 (see Rodríguez and Macías 2019) there has been improvements and extensions both in the solution and the MultEcore tool. The multilevel 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 made us to create an intermediate model for the insurance domain that was really not capturing any of the requirements stated in the challenge description. In the current version presented in this article (see Figure 1, this model has been avoided and this has helped us to demonstrate a flexibility aspect of MultEcore where the different domains do not need to have the same length as they are fully independent between them. 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. 2019b, Section 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 multilevel context. Thus, we have also used this construction to model the artefacts situation.

Improvements to our previous solution: - Gateway is now abstract. - Supplementary attributes.

8 Conclusions

In this paper, we have presented a solution to the Process Challenge proposed at MULTI 2019 workshop. Our multilevel modelling hierarchy has a total of five abstraction levels, two branches and 8 models, including the generic domain of process description and its refinement for the software engineering and the insurance domains. Each level is 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 the simulation of process execution through model transformations at the bottom levels. Our solution is based on the MultEcore tool and follows a conceptual framework which

enables EMF with the potential of becoming a multilevel modelling framework. This facilitates usage of the rich ecosystem of EMF in order to, for example, create such editors with Sirius and/or Xtext.

From a more conceptual standpoint, we believe that the focus that our approach has on flexibility and reusability allowed us to create an elegant, concise and correct multilevel hierarchy for the given domain of process modelling. We believe that this solution can be an interesting contribution for the challenge and be used to foster fruitful discussions within the MLM community.

References

Almeida J. P. A., Rutle A., Wimmer M., Kühne T. (2019) The MULTI Process Challenge. In: Burgueño L., Pretschner A., Voss S., Chaudron M., Kienzle J., Völter M., Gérard S., Zahedi M., Bousse E., Rensink A., Polack F., Engels G., Kappel G. (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 164–167

Almeida J. P. A., Rutle A., Wimmer M., Kühne T. (2021) The MULTI Process Challenge. In: Enterprise Modelling and Information Systems Architectures Available at https://bit.ly/3b3cQZV

Atkinson C., Gerbig R. (1st Jan. 2016a) 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. (1st Jan. 2016b) 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: Atkinson C., Grossmann G., Kühne T., de Lara J. (eds.) 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., Kühne T. (2001) The Essence of Multilevel Metamodeling. In: Gogolla M., Kobryn C. (eds.) «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings. Lecture Notes in Computer Science Vol. 2185. Springer, pp. 19–33

Atkinson C., Kühne T. (2005) Concepts for Comparing Modeling Tool Architectures. In: Briand L. C., Williams C. (eds.) 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

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

Committee T. M. W. (2018) The MULTI Process Challenge. In: CEUR Workshop Proceedings Available at https://bit.ly/3gKalPD

Department H. C. S. (n.d.[a]) MultEcore Website. https://ict.hvl.no/multecore/

Department H. C. S. (n.d.[b]) MULTI Process Challenge solution. https://github.com/alejandrort/no.hvl.multecore.examples.process2019

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 https://doi.org/10.1016/j.jlamp.2019.100497

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

Gamma E. (1995) Design patterns: elements of reusable object-oriented software. Pearson Education India

Henderson-Sellers B., Clark T., Gonzalez-Perez C. (2013) On the Search for a Level-Agnostic Modelling Language. In: Salinesi C., Norrie M. C., Pastor O. (eds.) 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: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 98–102

Jeusfeld M. A. (2019b) DeepTelos for ConceptBase: A Contribution to the MULTI Process Challenge. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 66–77

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level Modeling with Most General Instances. In: Comyn-Wattiau I., Tanaka K., Song I.-Y., Yamamoto S., Saeki M. (eds.) 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., pp. 673–682 http://ceur-ws.org/Vol-2245/multi%5C_paper%5C_5.pdf

Kühne T. (2018b) Exploring Potency. In: Wasowski A., Paige R. F., Haugen Ø. (eds.) Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018. ACM, pp. 2–12

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. LNCS Vol. 6141. Springer, pp. 1–20

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

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: MULTI@ MoDELS, pp. 66–75

Macías F., Rutle A., Stolz V. (2017) Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge. In: Proceedings of MULTI @ MODELS, pp. 269–273 http://ceur-ws.org/Vol-2019/multi%5C_9.pdf

Macías F., Wolter U., Rutle A., Durán F., Rodriguez-Echeverria 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

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., Durán F., Rutle A., Kristensen L. M. (2019b) 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: Proceedings of MULTI @ MODELS, pp. 152–163

Rodríguez A., Rutle A., Durán F., Kristensen L. M., Macías F. (2018) Multilevel modelling of coloured petri nets. 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. 663–672

Rodríguez A., Rutle A., Kristensen L. M., Durán F. (2019c) A Foundation for the Composition of Multilevel Domain-Specific Languages. In: MULTI@ MoDELS, pp. 88–97

Somogyi F. A., Mezei G., Urbán D., Theisz Z., Bácsi S., Palatinszky D. (2019) Multi-level Modeling with DMLA - A Contribution to the MULTI Process Challenge. In: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. 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: 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS Companion 2019, Munich, Germany, September 15-20, 2019. IEEE, pp. 134–141

Urbán D., Theisz Z., Mezei G. (2018) Self-describing Operations for Multi-level Metamodeling. In: Hammoudi S., Pires L. F., Selic B. (eds.) 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

A Complete model of Acme software engineering process

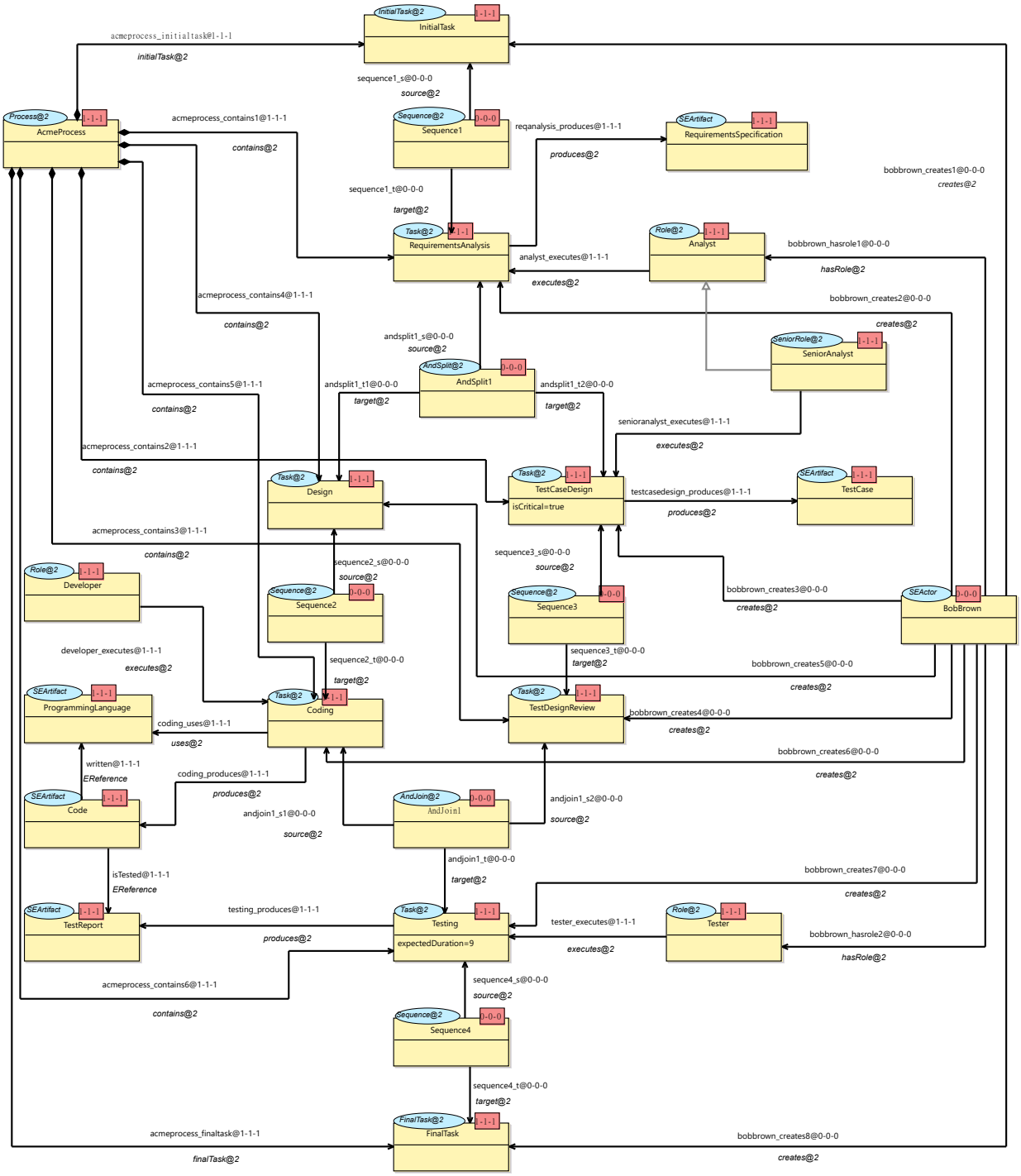

Figure 14: Level 3: Complete Acme software engineering process model

B Complete model of XSure insurance Claim Handling process

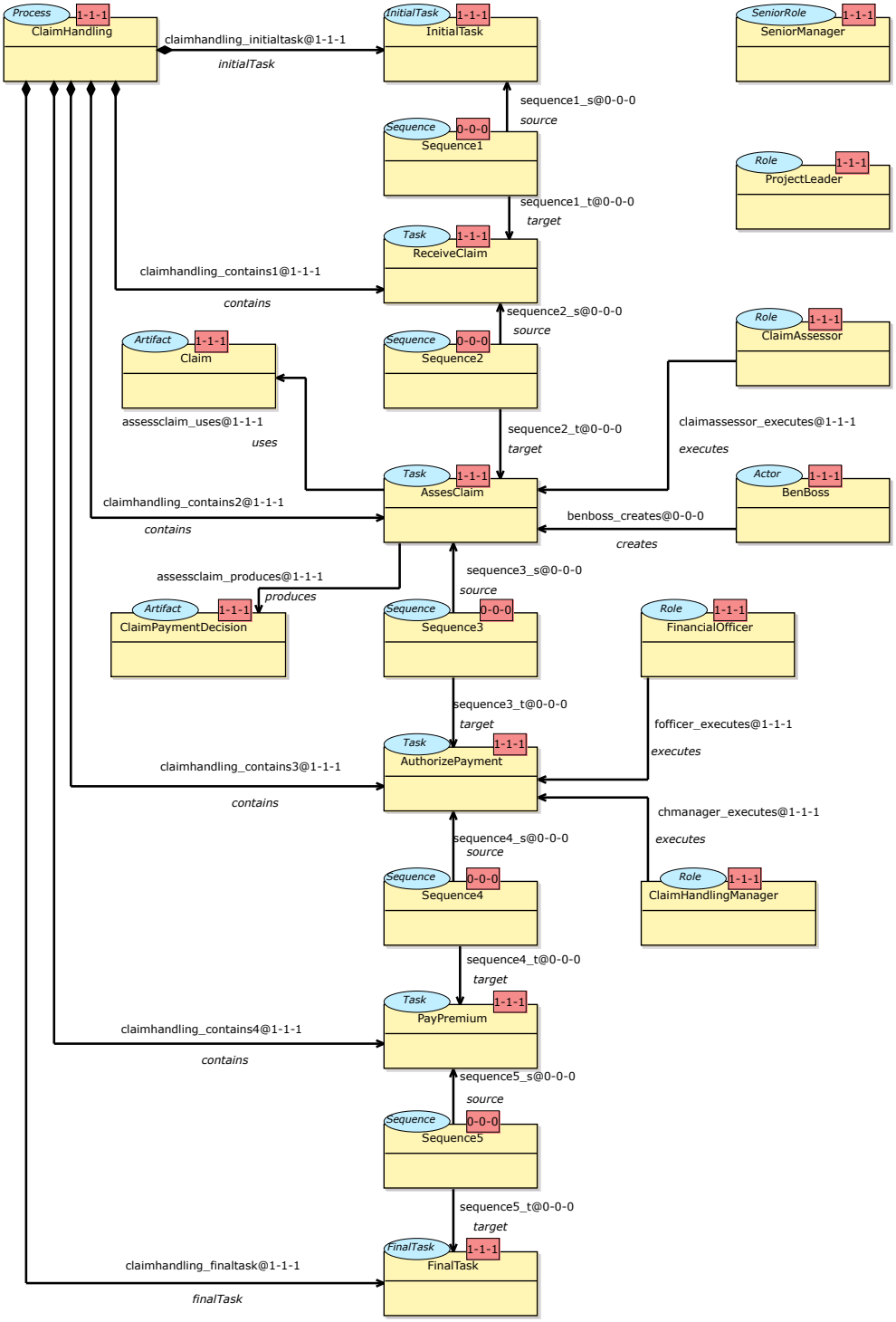

Figure 15: Level 2: XSure insurance Claim Handling process model