# Multilevel Modelling with MultEcore: A contribution to the MULTI Process challenge

Alejandro Rodríguez[1]
[1] *Department of Software Engineering, Sensor Networks and Engineering Computing*
[1]*Western Norway University of Applied Sciences*
Email: arte@hvl.no[1], fmac@hvl.no[2]
Bergen, Norway

Fernando Macías[1,2]
[2]*Department of Informatic and Telematic Systems*
[2]*University of Extremadura*
Email: fernandomacias@unex.es
Cáceres, 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.**

## I. 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 [1] to apply various MLM features which are key to be able to fulfil the criteria established for the MULTI Process challenge [2]. MultEcore enables multilevel modelling through the Eclipse Modelling Framework (EMF) [3], and therefore allows reusing the existing EMF tools and plugins. The MultEcore tool is available in [4] and the solution to this challenge can be downloaded in [5].

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* [6]). 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) [7], [8].

The key aspects that characterize our conceptual framework and have been applied to solve the challenge are summarized 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. II relevant aspects of the case presented in the challenge and in the context of the solution approach. Sect. III describes the complete scenario that contains the multilevel architecture built for this challenge. In Sect. IV 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 summarize and conclude the paper in Sect. V.

## II. Case analysis

The fourth paragraph in the introductory section of the challenge description already suggests that the domain has some common concepts that can be refined in lower levels of a hierarchy for different, more specific domains: "domain-specific concepts may be defined in their dedicated branches of a hierarchy of models without polluting the general terminology of process management". This approach aligns with the point of view that we use in other case studies for MultEcore (see, e.g. [7]), and we follow it for our solution. However, levels in MultEcore are used as an organisational tool that ensures a high level of flexibility and reusability, regarding both model definition, model maintainability and model transformation. Therefore, we do not necessarily align in every single case with the claim made in the first paragraph of the introduction to the challenge that "MLM allows for an arbitrary number of classification levels", since we do not argue that the typing relation in our multilevel hierarchy has always strict *classification* semantics, but the more broad *abstraction* semantics. That is, MultEcore uses levels as an organisational tool where the main rationale for locating elements in a particular level is grouping them by how abstract they are, and how reusable and useful they can be in that particular level. For example, our solution could be made more generic (outside the scope of the challenge) by replacing certain inheritance relations in our solution by typing, as with the AbstractRole, CombinedRole, Role and SeniorRole elements presented later in the paper. Still, we generally use typing relations with classification semantics, and the typing relation still implies that a node defines which attributes its instances can instantiate and which relations they can have to other nodes.

Thus, we can say that we enforce the *level cohesion* principle [9] as stated in the second point of Sect. 3.1 of the challenge description, but not *level segregation*. However, we are open to the possibility of introducing more *sanity checks* based on multilevel constraints for a specific domain if, for instance, we wanted to enforce strict classification. In this challenge, we rather focus on getting complete and concise models that can be easily used in a software engineering process. Currently, MultEcore ensures that typing relations cannot be circular, reversed, or inconsistent with supplementary typing. The interested reader can get more details about typing relations in MultEcore in [10].

This early analysis, together with the examination of the original Process case study, adapted in the challenge description from [11], lead us to the creation of a five-level, two-branch hierarchy, with levels numbered from 0 (on top) to 4 (in the bottom). In our approach to MLM, the top-most level is always occupied by a single, self-defining model of choice. We use Ecore for this purpose, mainly due to implementation reasons, since the MultEcore tool is designed as plugin for EMF. However, we choose not to overload our model presentation and description with all the details about this model, so the hierarchies and models depicted in this solution exclude level 0. 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 the authors in the original process case study (see Fig. 4 in [11]).

In order to support some of the requirements regarding the adaptation of the models to different languages and technological domains, we created supplementary hierarchies, which are a perfect fit to such kind of "aspect orientation" techniques for MLM. These are used to double-type the elements which need to be adapted without polluting the process-related concepts in the application hierarchy with them.

Regarding the completion of the case description, or the addition of requirements that our solution needs, we have not found the need to do so for our solution hierarchy. We do, however, include a few elements to represent secondary concepts that the challenge description does not name explicitly, but which make our models more explicit and flexible. These elements do not affect the general semantics of the models or the alignment with the requirements of the challenge. Hence, we discuss them in Sect. III as they appear in the models, and justify their addition in the context of the related elements that the challenge description mentions explicitly.

Finally, since the challenge description invites respondents to suggest "further requirements that clearly demonstrate the utility of multi-level modelling", we would like to take this opportunity to suggest the following for future editions:

- One could explore further. and more explicitly, how to translate the models into software, e.g. via code generation.

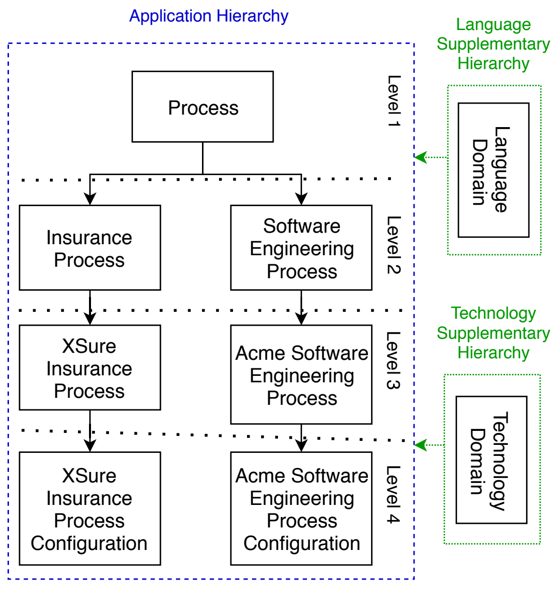
Application Hierarchy
Process
Level 1
Insurance Process
Software Engineering Process
Level 2
XSure Insurance Process
Acme Software Engineering Process
Level 3
XSure Insurance Process Configuration
Acme Software Engineering Process Configuration
Level 4
Language Supplementary Hierarchy
Language Domain
Technology Supplementary Hierarchy
Technology Domain

Fig. 1. Multilevel architecture constructed for the Process Challenge

- Loosely related to the previous point, instead of code generation, it could be interesting to extend the requirements to other model-related operations, especially model transformations and model querying. These techniques could be demonstrated for simulation or purposes or similar tasks.
- In our opinion, it would be beneficial if the challenge description provided a set of valid instances of the multilevel hierarchy of models. These could be model elements, probably without types, that should live at the bottommost levels of the different hierarchy branches, and could function as a sort of benchmark to check if a solution really has all the required concepts to specify such instances.

## III. Model presentation

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

We also describe the supplementary hierarchies (dotted boxes under Language Supplementary Hierarchy and Technology Supplementary Hierarchy) and the utilities they provide to overcome some of the requirements of the challenge. Note that supplementary hierarchies are not bound to a specific level but they are orthogonal to the application hierarchy. Lastly in this section we disclose constraint definitions using MCMTs. As we progress in the description of the hierarchy we will either indirectly go through the rules of the challenge description if no clarification is needed, or explicitly for those cases where some explanation is necessary. In both cases we reference the specific requirement is being fulfilled from the description ((**PX**) for for those applying to general processes and the insurance scope and (**SX**) for the software engineering domain). We only display the cardinalities on references in those cases where it is not the default one (0..*).

### *A. Level 1 - Process*

The first level contains the concepts concerning universal processes (see Fig. 2) and corresponds to the Process element in Fig. 1. This includes *process types*, *task types*, *artifact types*, and *actors*. The type of a node is indicated as a blue ellipse, e.g. EClass is the type of Process. The type of an arrow is written near the arrow in italic font type, e.g. EReference. The annotations in the rectangles at the top of the nodes, and after the names in the arrows (separated by '@') specify the potencies. In the case of MultEcore, a *potency* specification includes three values: the first two specify the first and the last levels where one can directly instantiate an element (min and max), and the third value, the number of times the element can be indirectly re-instantiated (depth). For attributes we drop the third value since they can only be instantiated once (i.e. depth is always 1). We also want to point out that we do not use the postfix *type* in the model.

The *composition* relation contains (with cardinality 1..*) between Process and Task models that a *process type* is defined by the composition of one or more *task types* (**P1**). A Task can have an expected duration. This is reflected in the attribute expectedDuration. We do not add any constraint checking that the expected duration is satisfied (with respect to begin and end dates) as this might not be respected by particular occurrences (**P8**). The potency in this case is 2-2, as this attribute is scoped for domain-specific task types (level 3). Tasks can also have a *begin date* and an *end date*. Attributes beginDate and endDate have 3-* as potency, meaning that they can be instantiated minimum three levels below (**P12**). Requirement **P9** indicates that a last attribute called isCritical is defined to specify that certain task types (in level 3) can be critical. The remaining of this requirement needs a constraint to ensure the correctness of it. Constraints are explained in Sect. III-D).

One might need to establish ordering constraints between *task types* by using Gateways, which can be Sequence, AndSplit, AndJoin, OrSplit or OrJoin (**P2**). This is captured via *inheritance* relationships between Gateway and the five possibilities. "A *process type* has one *initial task type* and one or more *final task types*"(**P3**). The references initialTask (with cardinality 1..1) and finalTask (with cardinality 1..*) relates Process with InitTask and FinalTask, respectively. They both inherits from Task.

*Actors* may have more than one *actor type* (**P15**). We define actor types as *roles*. The reference hasRole (with cardinality 0..n) between Actor and AbstractRole models such a requirement. We use for roles the traditional object-oriented Composite pattern [12]. We define AbstractRole as an abstract node (italic font in the name). On one hand, normal roles are defined as Role. SeniorRole node inherits from Role and it is defined to fulfill the requirement **P9**. We come back to this with an application in Sect. III-B2. On the other hand, we use CombinedRole to define roles than can be composed by simple roles (the 2..* cardinality ensures there are at least two roles combined). Also, *roles* (i.e., *actor types*) can have assigned task types whose instances can perform (**P5**). This is covered with executes reference from AbstractRole to Task. Each *actor* can both *create task types* and *perform tasks* (**P4** and **P6**). We model this with the references creates and performs from Actor to Task. Nonetheless, while the *creation* is concerned to *tasks types* (in level 3), the *performance* is related to *tasks* (in level 4). Thus, their potencies are different, 2-*-* for creates and 3-*-* for performs.

Another key point is that specific *actors* can belong to different abstraction levels. While, for instance, *Ben Boss* can *create* a *task type* (in level 3), *John Smith* (or even *Ben Boss* himself) could *perform* a task in level 4. As we do not support the use of cross-level relationships we have made the decision of being able to define Actor instances starting in level 2 (notice the 1-*-1 potency specification for Actor). We are aware that there might exist more than one instance of an actor among different levels of abstractions representing the same entity. However, this can be easily controlled as they would have the same node ID. The two references, produces and uses, from Task to Artifact capture that *tasks* can both *use* and *produce artifacts* (**P7**). Since we do not handle multiple

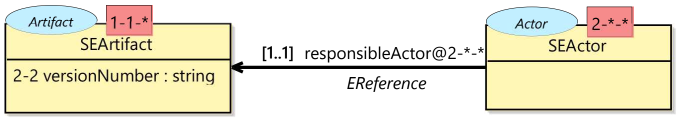

Fig. 3. Level 2: Software engineering process model

typing the requirement **P16** is not achieved. However, this can be done similarly as we have done with the *role* composition.

## *B. 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).

*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. For instance, the requirement **S10** states that "*software engineering artifacts* have a responsible *actor* and a *version number*. This applies to *requirements specification*, *code*, *test case*, *test report* but also any future types of *software engineering artifacts*". The introduction of this level forces to instances in levels below to carry such information. While SEArtifact must be instantiated in the level below (for instance, Acme artifacts such as CodeArtifact will have

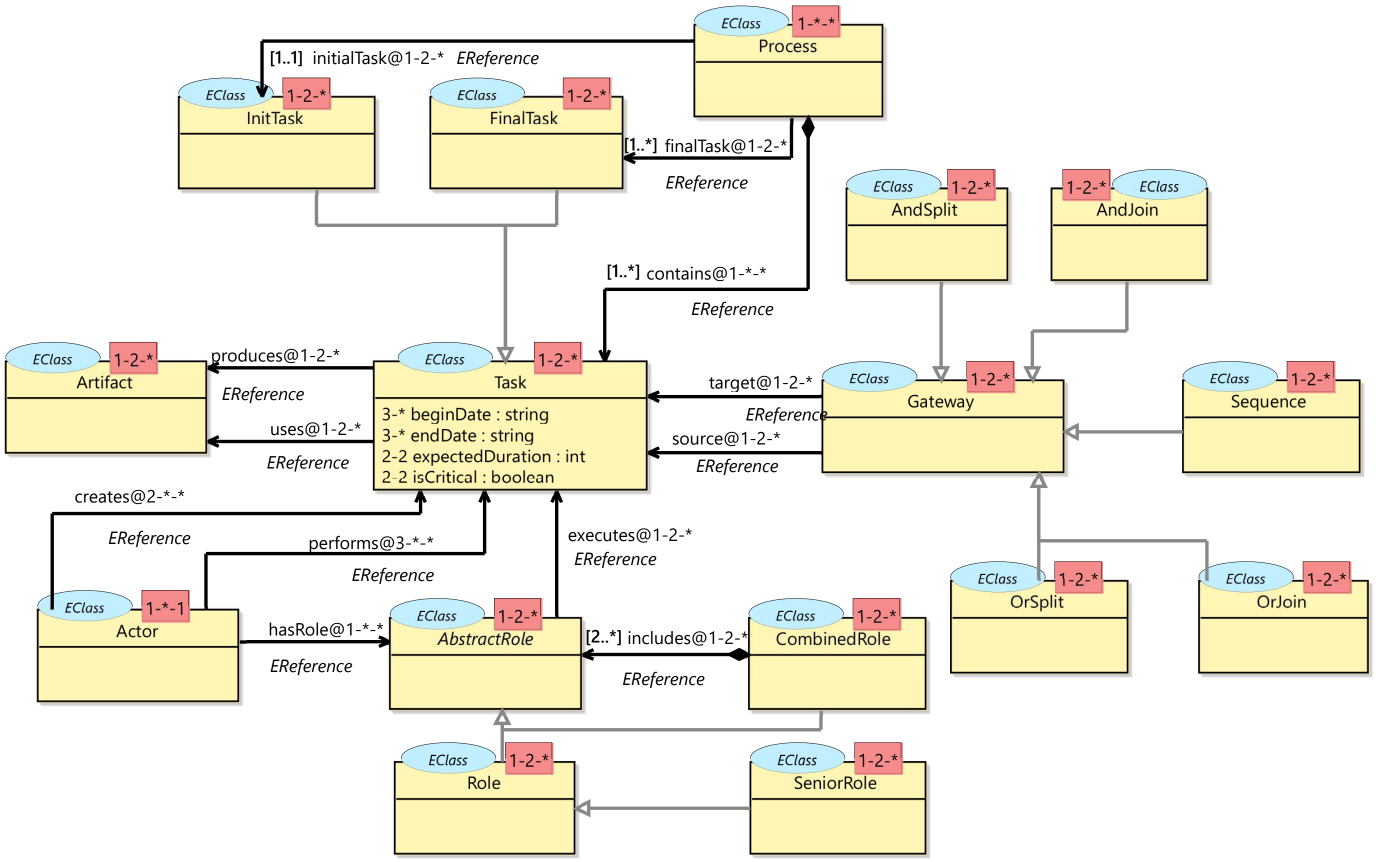

Fig. 2. Level 1: Process model

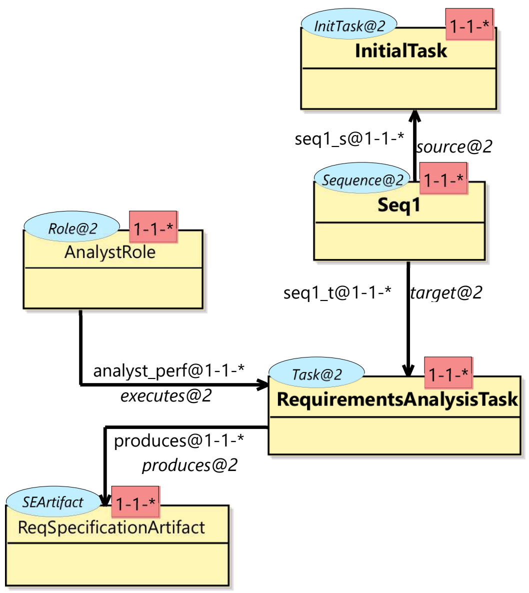

Fig. 4. Level 3: Fragment 1 of Acme software engineering process model

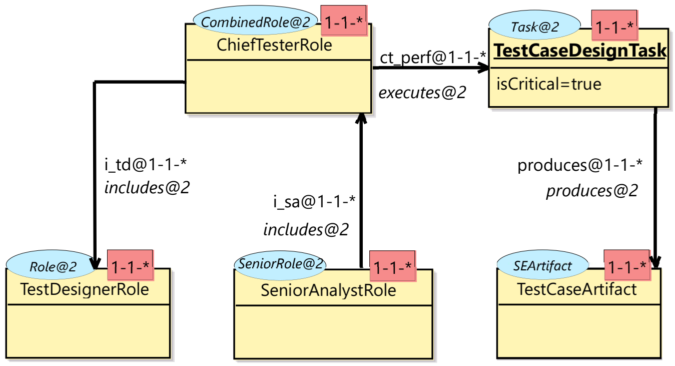

Fig. 5. Level 3: Fragment 2 of Acme software engineering process model

SEArtifact as type, in level 3), the relationship capturing that any software engineering artifact must have associated a responsible actor (cardinality 1..1 in responsibleActor reference) must be defined at least two levels below (in level 4, where one can define concrete Acme software engineering artifacts). Similarly, versionNumber attribute must be also instantiated two levels below. We do not provide more specification in this level, however, it can be further improved to add features that apply to every process within the software engineering domain.

*2) Level 3 - Acme software development 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. First, we examine aspects directly related to Fig. 1 of the challenge description. We have highlighted these elements with bold text.

In this level, it is specified the *types* that belong to the Acme software engineering domain. We define them as elements which type is Task@2 in level 1 (@2 indicates that Task is defined two levels above). For instance, we show in Fig. 4 that RequirementsAnalysisTask node represents *Requirements Analysis* not as a specific task but as a type. Since the model is quite big, we show excerpts of it to go through the different points. The complete model is located in Appendix A (Fig. 15).

DesignTask, TestCaseDesignTask, CodingTask, TestDesignReviewTask and TestingTask (displayed in Fig. 15) are represented in a similar way. An *initial task* and a *final task* are also reflected in InitialTask (which type is InitialTask@2, in the top side of Fig. 4) and FinalTask, respectively.

In the Acme diagram (Fig. 1 in [2]) one can see that tasks are connected by different *gateways*. We show in Fig. 4 Seq1 which is of type Sequence with InitialTask as source@2 (this reference is named seq1_s) and RequirementsAnalysisTask as target (named seq1_t). The rest of the gateways given in the challenge are shown in Fig. 15. E.g., AndSplit1 is an AndSplit gateway that splits RequirementsAnalysisTask (source@2) into DesignTask and TestCaseDesignTask via split1_t1 and split1_t2, respectively. Similarly, AndJoin1 joins CodingTask and TestDesignReviewTask into TestingTask.

Now we discuss some of the rules specified in Sect. 2.3 of the challenge document that apply to this level of abstraction. "A *requirements analysis* is *performed by* an *analyst* and *produces* a *requirements specification*" (**S1**). This is shown on the left side of Fig. 4 where the AnalystRole can perform RequirementsAnalysisTask (via analyst_perf reference of type executes@2) which produces ReqSpecificationArtifact (via produces with type produces@2).

"A *test case design* is *performed* by a *developer* or *test designer* and *produces test cases*" (**S2**). This requirement is strictly related to **S13**. It states that "*designing test cases* is a *critical task* which must be performed by a *senior analyst*. Here we make use of the composite pattern specified for roles in Fig. 2". This is shown in Fig. 5 where a CombinedRole@2 named ChiefTesterRole is composed by TestDesignerRole of type Role@2 and SeniorAnalystRole of type SeniorRole@2. Note that isCritical is set as true in TestCaseDesignTask, and only a ChiefTesterRole is allowed to perform (ct_perf) it. This analogously applies to ChiefDeveloperRole (which is a combination of DeveloperRole and SeniorDeveloper). Due to similarities we do not make the construction for *developer* role. To complete **S2**, TestCaseDesignTask produces TestCaseArtifact.

"An occurrence of *coding* is *performed* by a *developer* and *produces code*" (**S3**). Fig 6 shows a fragment of the model where this property is reflected. DeveloperRole is connected via dev_perf to CodingTask which uses ProgLangArtifact. "Instances of *coding* must furthermore reference one or more *programming languages* employed" (**S3**). This is constrained structurally in the model. Note in Fig 6 that the cardinality of the reference uses between CodingTask and ProgLangArtifact is 1..*. One must always create an instance of ProgLangArtifact connected to an instance of CodingTask in case of the latter is generated.

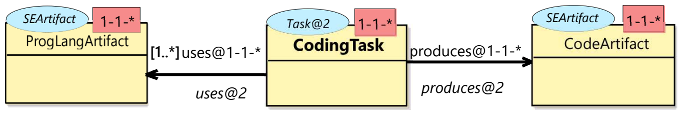

Fig. 6. Level 3: Fragment 3 of Acme software engineering process model

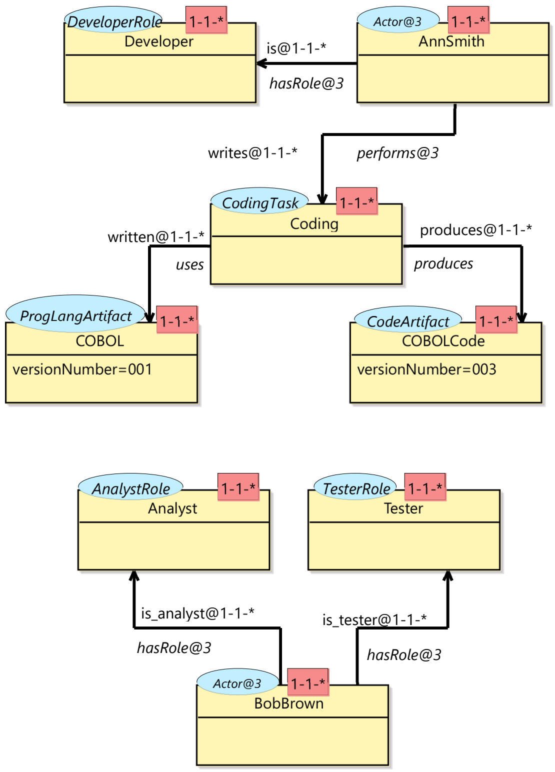

Fig. 7. Level 4: Acme software engineering process configuration model

Notice in the bottom left part of Fig. 15 (Appendix A) the attribute expectedDuration has been instantiated to 9 for TestingTask (**S12**). To demonstrate "*Bob Brown* has created all *task types* in this *software engineering process*" (**S11**) we provide two examples in the bottom right corner of Fig. 15, where BobBrown creates FinalTask and TestDesignReviewTask. We do not show the rest of references to the other tasks as they would be rather similar to the ones just mentioned.

*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. This is shown in the model depicted in Fig. 7 and it corresponds to the Acme Software Engineering Process Configuration element in Fig. 1.

Observe in Fig. 7 that Coding uses COBOL as artifact and produces COBOLCode (**S5** and **S6**). As specified in **S10**, all *software engineering artifacts* must have a *version number*. Notice that in both COBOL and COBOLCode the attribute versionNumber is instantiated.

At the top of Fig. 7, AnnSmith, who is a Developer, performs (via writes reference) Coding which has CodingTask as type (**S7**). At the bottom of Fig. 7 we show how an actor BobBrown may have multiple roles, Analyst and Tester in this case (**S11**). This is one example of the situation described at the end of Sect. III-A. BobBrown has been defined in level 3, to model that he has created all task type of software engineering artifacts, but also that BobBrown has the aforementioned roles.

## C. Insurance process domain

The challenge description [2] provides an optional incorporation for the insurance domain which we have also modelled to demonstrate flexibility of MultEcore. We have constructed the models (level 2, 3 and 4) from the examples provided in Sect. 2.2 of [2].

*1) Level 2 - Insurance process:* This level is not explicitly extracted from the challenge document. However, and similarly as with the software engineering branch, we define this level to capture elements that might affect to any insurance company. It is represented in Fig. 8 and it corresponds to

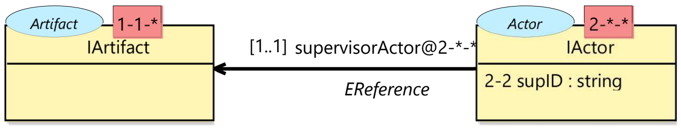

Fig. 8. Level 2: Insurance process model

Insurance Process in 1. For instance, we have establish that insurance artifacts (IArtifact node) must have a supervisor (supervisorActor reference). IActor are insurance actors that can supervise insurance artifacts, and must have a supervisor ID (supID attribute). Note that this relationship must be conformed two levels below, where one can talk about specific artifacts. Therefore, IActor must be instantiated, at least, in level 4 (potency 2-*-*), and so the supervisorActor (potency 2-*-*) reference and supID (potency 2-2). We show an application of such an abstraction in Sect. III-C3.

*2) Level 3 - XSure insurance process:* The insurance elements that are mentioned in the challenge document are provided as fragments (in comparison with the software engineering domain where it is presented as a sequence model). Thus, in this subsection, we go through the elements concerning level 3 as they appear in the **PX** requisites. The model in such a level corresponds to XSure Insurance Process element in 1. One can find the complete model in B (Fig. 16).

A *claim handling* process is defined by the composition of *receive claim*, *assess claim*, *text* and *authorize payment* task types (**P1**). We do not define specific gateway type for this domain as it has been already shown in III-B2. The same applies to *initial/final tasks*.

"*Ben Boss creates* the task type *assess claim*" (**P4**). This is depicted at the top of Fig. 9 where BenBoss creates AssesClaimTask. "Only *claim handling manager* or a *financial officer* can *authorize payments*" (**P5**). Such a requirement is illustrated at the bottom of Fig. 9, AuthorizePaymentTask is executed either by FinancialOfficerRole or ClaimHandlingManagerRole. We also capture that "*Assess claim* uses a *claim* and *produces* a *claim payment decision*" (**P7**).

The capability of using inheritance in both models and MCMTs (which we use to specify constraints), we can define that if a *Manager* is allowed to perform, for instance, a *review claim*, then a *Senior Manager* is also allowed **P18** to do so. One can see in Fig. 10 that ReviewClaimTask can be performed (executes reference) by either ManagerRole or SeniorManagerRole (as it inherits from ManagerRole). We come back to this requirement in the next subsection where we show specific examples of the insurance domain (level 3).

*3) Level 4 - XSure insurance process configuration:* Some of the suggestions for the insurance domain belong to a concrete configuration of the XSure company. This level corresponds to XSure Insurance Progress Configuration element in Fig. 1.

"We might have that *John Smith* and *Paul Alter* are the only *actors* that can *asses claims* (**P6**)". In Fig. 11 we show how this is structurally specified. Besides, this requires a constraint that checks whether each instance of AssessClaimTask is performed by either JohnSmith or PaulAlter. We detail constraints in Sect. III-D. "*Assessing Claim 123* has a *begin date 01-Jan-19* and *end date 02-Jan-19*" (**P12**). This is captured in the instantiated attributes of AssessingClaim123 node.

We also show that an *actor*, for example, "*John Smith* may have more than one *actor type* (role in our domain), e.g., *senior manager* and *project leader*" (**P15**). As commented at the end of Sect. III-C2, if a *manager* is allowed to perform certain tasks, so it is a *senior manager*. This is reflected Fig. 11 where JohnSmith has the role SeniorManager (of type SeniorManagerRole) performs the ReviewingClaim123 which type is ReviewClaimTask. Notice that we have these types defined in level 3 (Fig. 10), and that SeniorManagerRole should be able to perform tasks of type ReviewClaimTask. Again, we need a constraint to make sure that first, an actor with the correct role can perform the task, and second, that elements inheriting from the super class can also perform the operations.

On the right side of Fig. 11 we display an application of the concepts defined in the level 2 (commented in Sect. III-C1). An insurance actor JackMiller (with supID = 0004)supervises an artifact named ClaimPayment.

*4) Supplementary hierarchies:* In this subsection we discuss how we make use of one of the key features that characterizes MultEcore. This is motivated by the requirement **P19**: "*Artifacts*, *actors* and all *types* (including *process*, *task*, *artifact* and *actor types*) are given a set of alternative *names* (e.g., to cope with internationalization requirements or variation in terminology)".

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 two supplementary hierarchies that can provide different naming dimensions to the elements along the application hierarchy. For instance, we define one supplementary hierarchy for languages and one for technologies name spaces. Initially, the supplementary hierarchies defined contains, each of them, one model, and one node. It is worth to remind that elements from supplementary hierarchies can be used in an orthogonal manner. This means that one can use such elements and instantiate their attributes in any level of our application hierarchy. From this point, we

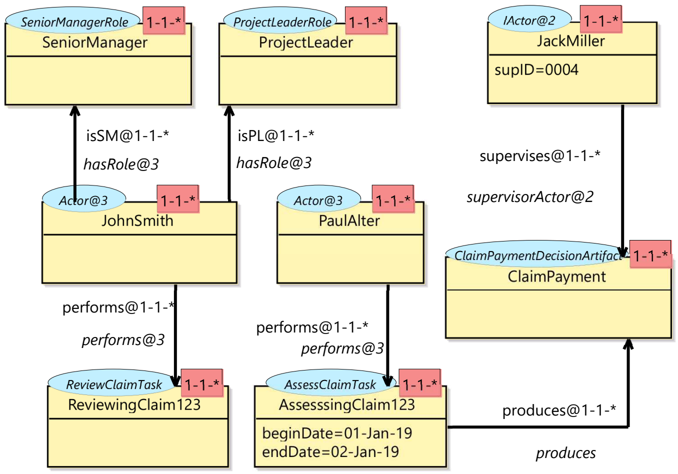

Fig. 11. Level 4: XSure insurance process configuration model

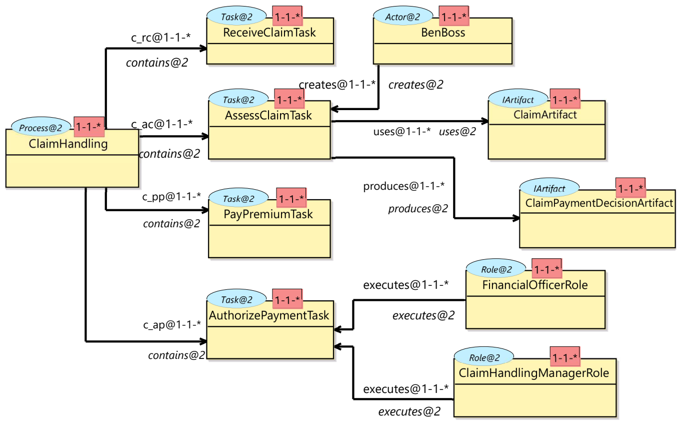

Fig. 9. Level 3: Fragment 1 of the XSure insurance process model

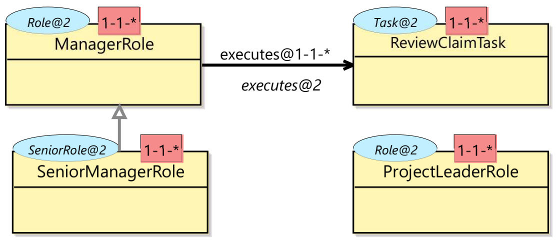

Fig. 10. Level 3: Fragment 1 of the XSure insurance process model

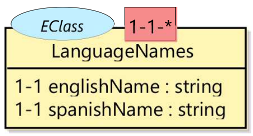

Fig. 12. LanguageNames Supplementary typing

use the *language* hierarchy as example for the explanation. Fig. 12 displays the LanguageNames node which contains two attributes, englishName for English and spanishName for Spanish. These attributes can be instantiated in any node that includes LanguageNames as supplementary type. An example is shown in Fig. 13 where ReviewClaim123 includes LanguageNames (green ellipse at the right side of the class) as supplementary type and instantiates the attributes coming from the supplementary dimension (englishName = Reviewing claim 123 and spanishName = Revisar Reclamacion 123). The advantages of using this approach is that further languages can be added and used. Furthermore,

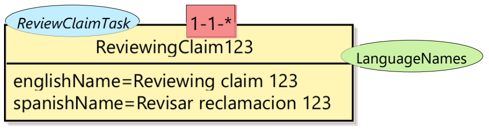
ReviewClaimTask
1-1-*
ReviewingClaim123
LanguageNames
englishName=Reviewing claim 123
spanishName=Revisar reclamacion 123

Fig. 13. ReviewClaim123 node

we can have several supplementary hierarchies, for instance, one could include the *technology* supplementary hierarchy and add new features without harming previous work. This permits to have several attributes altogether within the same node (e.g., ReviewingClaim123), belonging to different dimensions.

## D. Cross-level constraints

Multilevel Coupled Model Transformations (MCMTs) have already been proposed for the definition of behaviour descriptions (i.e., dynamic semantics) [13]. 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 [8]. However, the specification and execution of static semantics, i.e., constraints to check some structural and semantic correctness is part of our current work.

Transformation rules can be used to represent actions that may happen in the system. Conventional in-place 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. One of the possibilities to define constraints we are currently investigating is to be able to find a correspondence in the models through a two-step technique. 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

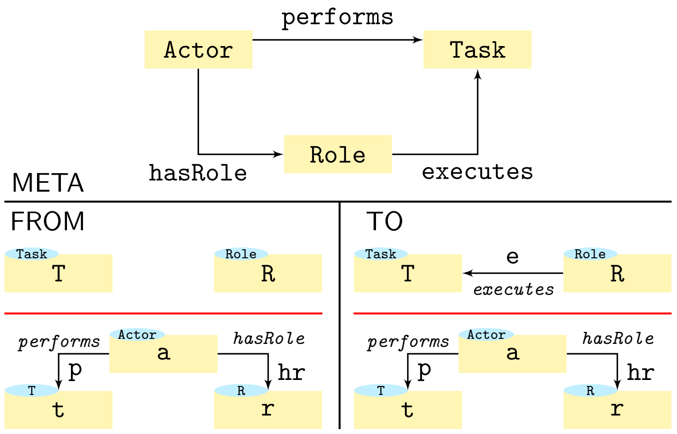

Fig. 14. Constraint satisfying requirement **P17**

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 that performs a task must be authorized for that task. Typically, a class of actors is automatically authorized for certain classes of tasks." Fig. 14 shows a MCMT 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). The FROM and TO blocks are in this case composed by two different level of abstraction (separated by red horizontal lines). It is worth to mention that the three levels specified in this rule are not required to be consecutive. Consequently, the level in the META block in this constraint matches the level 1 of the application hierarchy (the process level) and the two levels appearing in the FROM and TO blocks match levels 3 and 4, respectively. The match is valid for both insurance and software engineering 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 subsets of the hierarchy 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 artifacts to be used for performing model repair.

At the META level, we mirror part of *process* metamodel, defining elements like Actor, Task and performs that are used directly as types in the levels below in the rule. These elements are defined as constants, meaning that the name of the pattern element must match an element with the same name in the typing chain. 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 to an instance of R (r) that must be at the same time an instance of Role.

Translated to the rule, if an actor (a) with a certain role (r) performs (p) a task (t) (FROM match), then, the type of that role R must be allowed to perform (e of type executes) such kind of tasks (T) (TO match). For instance this constraint would ensure the requirement **P5**.

Even though we are already working on implementing this functionalities in the MCMTs, this challenge has served us to find different cases and it has provided us with new case studies to test them.

## IV. Discussion

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 [10], including the first edition of the bicycle challenge [14].

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. 2) 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 [15] and [10].

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 [16] 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 [17] and [18].
- 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 [2]] *and demonstrate the use of multi-level features?*" We believe that our solution contains all the required concepts and constructions required in the challenge description. In most cases, these constructions do not require workarounds or additional concepts, and we discuss and justify our choices in the few cases where we need them. Furthermore, our solution prominently makes use of multiple levels, three-valued potency specification and double typing (through 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 [2]]*?*" The main part of this section is dedicated precisely to the discussion of those criteria, in the same order that they are enumerated in [2], 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.

## V. 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

[1] F. Macías, A. Rutle, and V. Stolz, "Multecore: Combining the best of fixed-level and multilevel metamodelling." in *MULTI@ MoDELS*, 2016, pp. 66–75.

[2] J. P. A. Almeida, A. Rutle, M. Wimmer, and T. Kühne, "The MULTI Process Challenge," *MULTI @MODELS*, 2019, available at https://bit.ly/2JeDEYi.

[3] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[4] H. C. S. Department, "MultEcore Website." [Online]. Available: https://ict.hvl.no/multecore/

[5] ——, "MULTI Process Challenge solution." [Online]. Available: https://github.com/alejandrort/no.hvl.multecore.examples.process2019

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

[7] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria, "Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour," *Journal of Logical and Algebraic Methods in Programming*, 2019.

[8] A. Rodríguez, F. Durán, A. Rutle, and L. M. Kristensen, "Executing Multilevel Domain-Specific Models in Maude," *Journal of Object Technology*, vol. 18, no. 2, pp. 4:1–21, 2019.

[9] T. Kühne, "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.*, 2018, pp. 673–682. [Online]. Available: http://ceur-ws.org/Vol-2245/multi_paper_5.pdf

[10] F. Macías, "Multilevel modelling and domain-specific languages," PhD thesis, Western Norway University of Applied Sciences and University of Oslo, 2019.

[11] J. D. Lara and E. Guerra, "Refactoring Multi-Level Models," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 4, p. 17, 2018.

[12] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about Maude a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[14] F. Macías, A. Rutle, and V. Stolz, "Multilevel modelling with multecore: A contribution to the MULTI 2017 challenge," in *Proceedings of MULTI @ MODELS*, 2017, pp. 269–273. [Online]. Available: http://ceur-ws.org/Vol-2019/multi_9.pdf

[15] U. Wolter, F. Macías, and A. Rutle, "On the Category of Graph Chains and Graph Chain Morphisms," University of Bergen, Department of Informatics, Tech. Rep. 2017-416, March 2017.

[16] U. Frank, "Multilevel modeling - toward a new paradigm of conceptual modeling and information systems design," *Business & Information Systems Engineering*, vol. 6, no. 6, pp. 319–337, 2014.

[17] C. Atkinson and R. Gerbig, "Flexible deep modeling with Melanee," in *Modellierung 2016*, ser. LNI, S. Betz and U. Reimer, Eds., vol. 255. Bonn: Gesellschaft für Informatik, 2016, pp. 117–122.

[18] J. de Lara and E. Guerra, "Deep meta-modelling with MetaDepth," in *Objects, Models, Components, Patterns*, ser. LNCS, vol. 6141. Springer, Jul. 2010, pp. 1–20.

## Appendix

### A. *Complete model of Acme software engineering process*

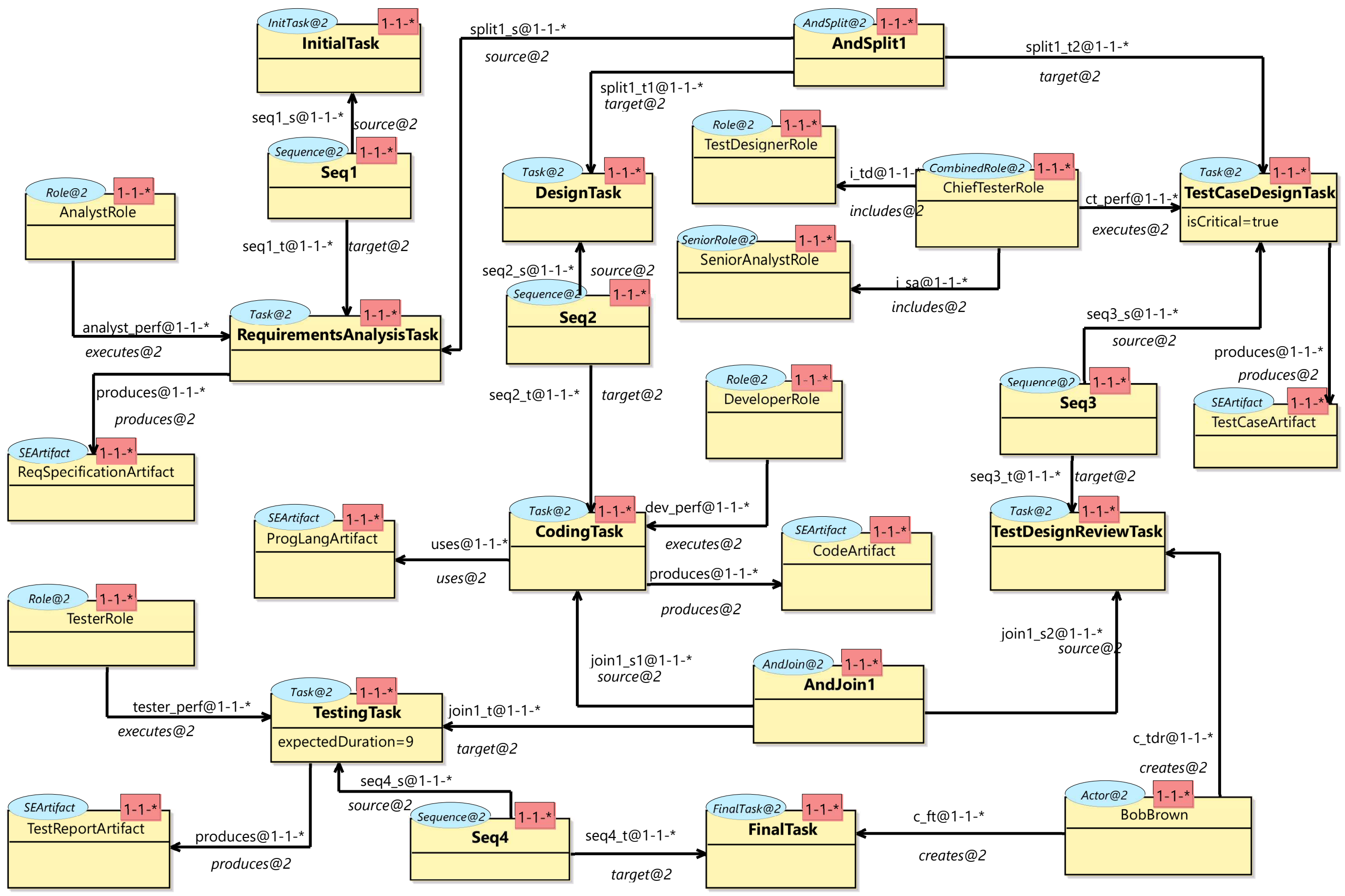

Fig. 15. Level 3: Acme software engineering process model

## B. Complete model of XSure insurance process

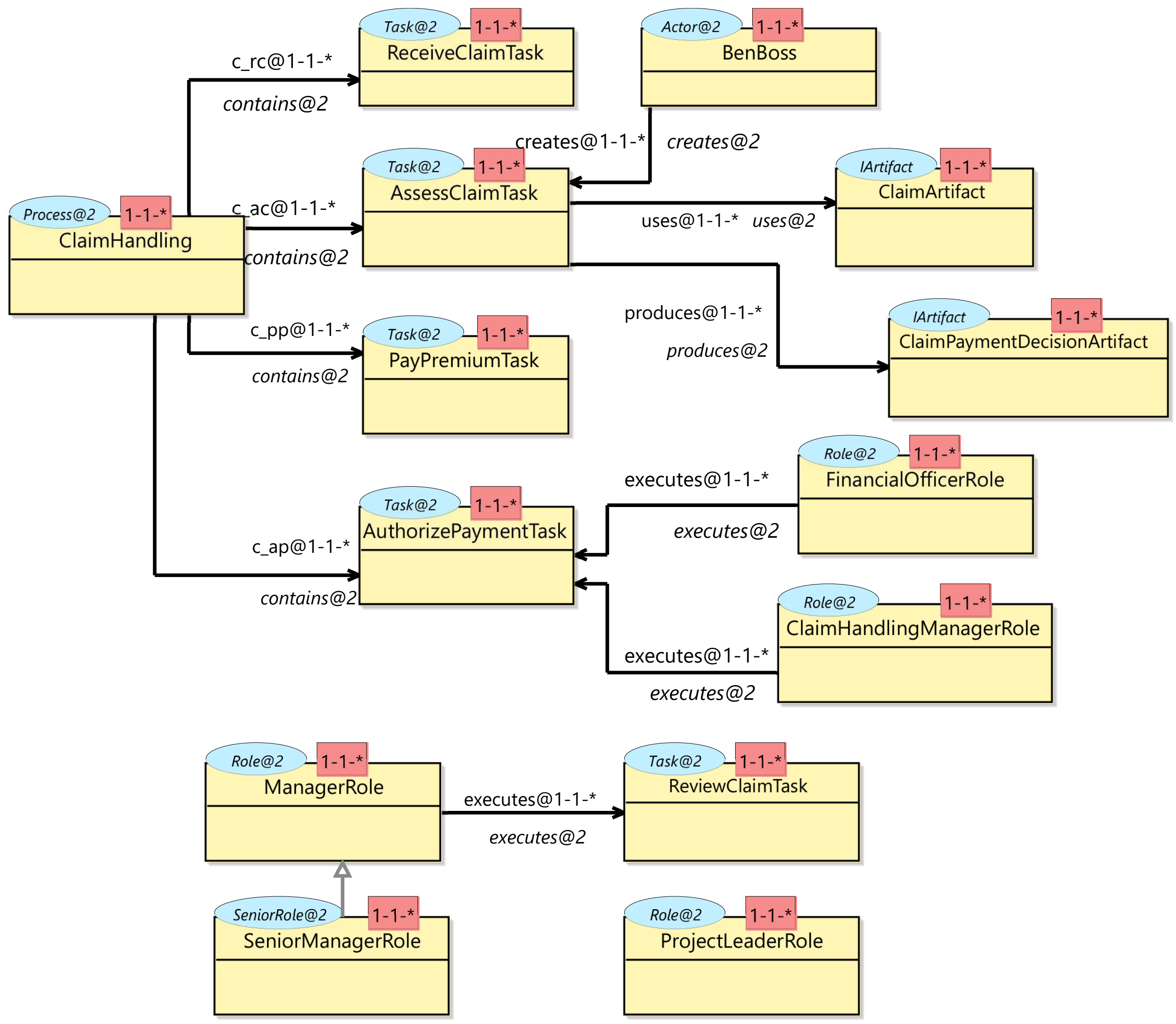

Fig. 16. Level 3: XSure insurance process model