Multi-Level Modeling with Openflexo/FML

A Contribution to the Multi-Level Process Challenge

Sylvain Guérin[a], Joel Champeau[a], Jean-Christophe Bach[*,b], Antoine Beugnard[b], Fabien Dagnat[b], Salvador Martínez[b]

[a] ENSTA Bretagne, Lab-STICC, UMR 6285, Brest, France
[b] IMT Atlantique, Lab-STICC, UMR 6285, Brest, France

Abstract. *Model federation is a multi-model* management *approach based on the use of virtual models and loosely coupled links. The models in a federation remain autonomous and represented in their original technological spaces whereas virtual models and links (which are not level bounded) serve as control components used to present different views to the users and maintain synchronization. In this paper we tackle the* EMISAJ multi-level process modeling challenge*, which consists in providing a solution to the problem of specifying and enacting processes. Solutions must fulfill a number of requirements for a process representation defined at an abstract process-definition level and at various more concrete domain-specific levels, resulting in a multi-level hierarchy of related models. We present a solution based on model federation and discuss the advantages and limitations of using this approach for multi-level modeling. Concretely, we use virtual models and more precisely the Federation Modeling Language (FML) that serves to describe them as the main building block in order to solve the process modeling challenge whereas the federation feature is used as a means to provide editing tools for the resulting process language. Our solution fulfills all the challenge requirements and is fully implemented with the Openflexo framework.*

Keywords. Model federation • Multi-Level Modeling • System modeling languages • Abstraction, modeling and modularity • Reusability

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

1 Introduction

Model-driven engineering (MDE) has traditionally adopted a strict hierarchical two-level approach, where metamodels reside in a certain meta-level, and models are created one meta-level below by using types from the metamodel. Notable examples of this approach are the widespread Eclipse Modeling Framework (EMF) (Steinberg et al. 2008) and the Object Management Group Meta-Object Facility (MOF) (OMG 2013).

This strict approach shows its limitations when modeling complex domains requiring more than one level of specialization, e. g., to adapt the model to application subdomains. Indeed, the two-level approach fails to acknowledge and support: 1) the existence of different forms of classification and/or instantiation (e. g., ontological vs linguistics); and 2) the duality type-object for model elements. Therefore, while it may still be used for the aforementioned complex modeling tasks, the strict approach entails the introduction of accidental complexity in both the modeling process and the resulting modeling artifacts.

In order to tackle this problem, the *multi-level* modeling paradigm has been introduced. Contrary to the strict approach, multi-level modeling advocates the use of a flexible number of levels as

* Corresponding author.
E-mail. jc.bach@imt-atlantique.fr

well as more flexible relations between them. This paradigm is gaining traction within the modeling community as evidenced by the contribution of many new multi-level modeling approaches and tools such as Melanee (Atkinson and Gerbig 2016), LIMM (Golra and Dagnat 2011), MetaDepth (De Lara and Guerra 2010), MultEcore (Macías et al. 2016), DeepTelos (Jeusfeld and Neumayr 2016) or DMLA (Urbán et al. 2017). Taking advantage of this vibrant state of affairs, and in order to foster discussion and enable comparison between competing approaches, a multi-level modeling challenge has been created for the MULTI workshop. This paper is a response to the latest multi-level modeling challenge (Almeida et al. 2021) which consists in providing a solution to the problem of specifying and enacting processes. Solutions to the *Multi-level Process Challenge* must fulfill a number of requirements for a process representation defined at an abstract process-definition level and at various more concrete domain-specific levels, resulting in a multi-level hierarchy of related models. In this article, we respond to this challenge using a solution based on model federation (Golra et al. 2016a).

Model federation is a multi-model management approach based on the use of virtual models and loosely coupled links. The models in a federation remain autonomous and are represented in their original technological spaces whereas virtual models (also called conceptual models) and links serve as control components used to present different views to the users and maintain synchronization. Our solution is based on the model federation infrastructure. Concretely, we use Openflexo and its internal Federation Modeling Language (FML), a language to create, link and manage virtual models, in order to solve the process challenge while the federation feature is used solely so as to provide tools for the resulting process language. Our solution, which is fully implemented and executable, meets all the requirements.

Our modeling approach is based on a language (FML) which provides the (linguistics) concepts that are used to define the multi-level models and meta-models that are needed. FML allows defining models as first class entities (which we call *virtual models*) and linking them together (by extension). In the Multi-Level Process Challenge case study, we exploit this possibility to:

- Define, extend and adapt models, keeping the history, to meet the modeling requirements of the case study,
- Align this hierarchy of models with requirements evolution (P1–P19 then S1–S13).

This methodological choice allows us to adapt the models and keep track of their evolution each time the requirements evolve.

Contrary to a classical multi-level approach, in which the notion of multi-level is integrated in the language and must therefore be respected, our approach allows building multiple levels on demand, and with the appropriate form. Without multilevel-specific operators/concepts, the "custom" construction must be done explicitly during the model development and evolution process.

Our solution to the challenge is therefore not a response using a classical multi-level approach that would allow an efficiency comparison with another classical multi-level solutions, but an approach that proposes another organization of multi-level hierarchy through the translation of needs into levels over time and the possibility to build an evolving hierarchy of adapted models. This is made possible by (1) the reification of the notion of a model and (2) the possibility to interconnect models (specialization).

The rest of the paper is organized as follows. Sect. 2 presents our approach and the technology it relies on. Sect. 3 is the analysis of the problem. We detail our model in Sect. 4 and show how it satisfies the challenge requirements in Sect. 5. We assess the modeling solution in Sect. 6. Sect. 7 presents the related work before the conclusions.

2 Technology

To meet the Multi-Level Process Challenge, we have decided to use the language infrastructure of model federation approach (Golra et al. 2016a). *Model federation* is a way to assemble models

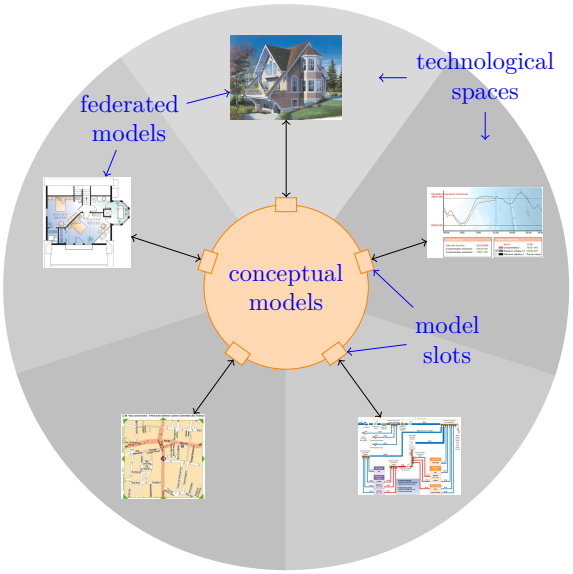

Figure 1: The model federation approach

using some kind of low-coupling links. It was initially formulated to respond to the OMG's RFP Semantic Modeling for Information Federation (Object Management Group 2016). In contrast to approaches that compose metamodels into a single large metamodel grouping all needed entities, model federation build links between models and metamodels (even through levels) to make "things" work together. As an illustration of this feature, we developed a free-modeling editor presented in (Golra et al. 2016b) that frees the designer from the bonds of model/metamodel conformity. Another notable feature of this approach is the strong decoupling among tools that remain usable after federations are made. So to provide an adaptable framework, we have developed a flexible modeling language offering modeling features without strict modeling levels. In a federation, the targeted modeling approaches can be unknown at the beginning and also can integrate several modeling levels.

We decided to use this approach since it offers the possibility to link and to navigate among levels. Before we describe the architecture in next section, here are some key concepts implemented in the Openflexo (Openflexo 2019) framework.

This framework relies on the architecture of Fig. 1. A federation gathers a set of conceptual models, named *virtual models* and a set of *federated models*. Each federated model pertains to a *technological space* and uses the language of its specific paradigm while a virtual model is built using the Federation Modeling Language (FML). Each federated model is an autonomous element that may evolve with its own tools. The virtual models serve as control elements binding the federated models together. In this paper, we mainly use conceptual models, and more precisely, FML, its domain specific modeling language. The only use of the federation feature is on the toolset for the process language.

A UML-like representation of a simplified metamodel of FML is provided in Fig. 2. FML is a language designed to define virtual models. A *virtual model* is composed of a set of *concepts* (named FlexoConcept in Fig. 2), while itself being a *concept*. Hence, *virtual models* are structuring units forming architectures, while *concepts* are the core entities. A concept has a set of *properties* (FlexoProperty) and *behaviors* (FlexoBehaviour). *Properties* can be either simple variables, roles (pointers to a modeling element in a *virtual model* or an external technology-specific model), or properties bound to complex control graphs. FML is designed to define not only the structure of virtual models but also the collection of actions that can be performed on them. These actions are called *behaviors* and rely on behavioral primitives called EditionAction. These *behaviors* can either be called or triggered by events. The reactive behaviors are useful when a federated model evolution needs to trigger a computation. Note that we do not exploit this possibility in the challenge.

A parallel to the object-oriented approach is useful to understand FML[1]. A concept corresponds to a class, its properties to the attributes of the class and its behaviors to the methods of the class. These properties have types defining the kind of value the role will point to at runtime. Whenever an element external to the federation space is used, one needs to use a *model slot*. A model slot is

[1] Some features of FML do not exist in the object-oriented approach.

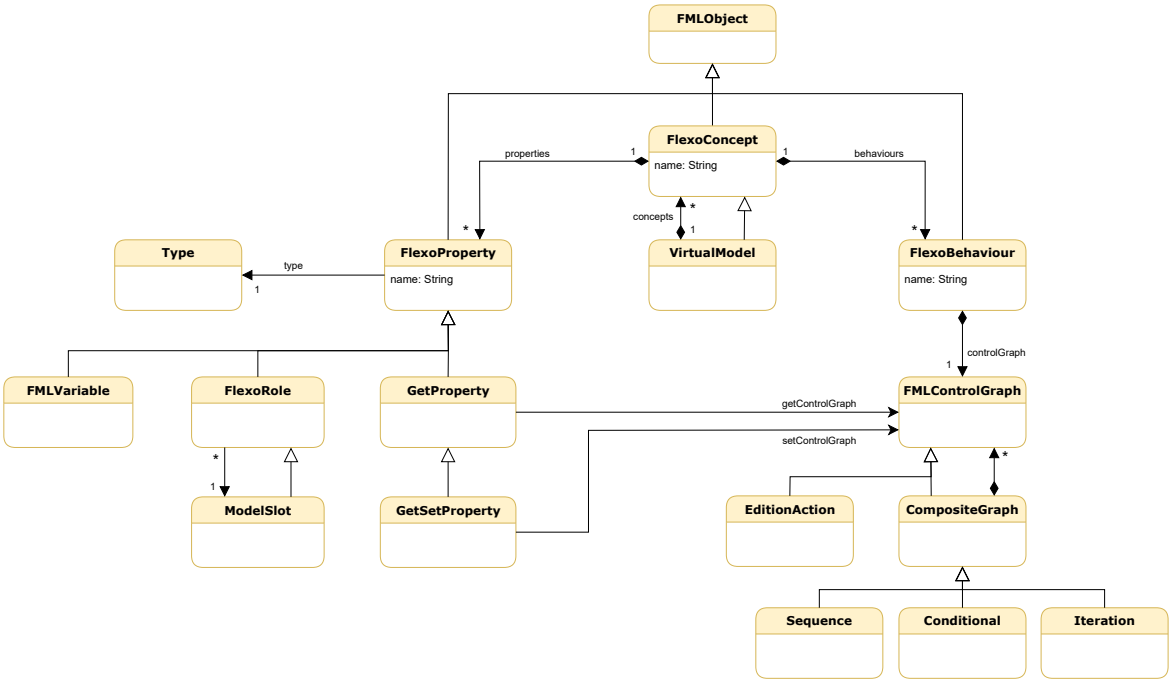

Figure 2: Representation of the main concepts of FML language

a mediation entity in charge of giving access to external elements using a *technology adapter*[2] .

When the FML execution engine runs a federation, it creates virtual model instances containing concept instances. Some concept instances are connected to external elements through model slot instances. To illustrate this, Fig. 3 shows how to realize a small subset of BPMN with Openflexo/FML. Concretely, we have designed a *virtual model* (`SimplifiedBPMN`) with 3 *concepts* representing a process and its contained tasks. An instance of this *virtual model* is presented in the bottom of the figure, showing instances with their instanceOf relationships with their model. The following listing shows the FML code of this *virtual model*. Note that the composition relation of Fig. 3 between `BPMNProcess` and `BPMNTask` is realized in the FML code by concept containment. Whenever an instance of `BPMNTask` is created, it must be in a container (instance of `BPMNProcess`) and the two instances get linked together.

```
// A simplified BPMN metamodel exposing the
// concepts BPMNProcess and BPMNTask
model SimplifiedBPMN {
  // Abstract concept BPMNElement defining a name
  abstract concept BPMNElement {
    String name;
    // A basic constructor
    create(String name) { this.name = name; }
  }
  concept BPMNProcess extends BPMNElement {
    // A basic constructor
    create(String name) { super(name); }
    // Other properties and behaviours of the
    // BPMNProcess concept
    ...
    concept BPMNTask extends BPMNElement {
      // A basic constructor
      create(String name) { super(name); }
      // Encodes execution of task
      execute() {
        ...
      }
      // Other properties and behaviours of the
      // BPMNTask concept
    }
    // Other concepts
  }
}
```

An open source tool, Openflexo,[3] supports

[2] It is a reusable library that defines connections between the FML execution engine and a particular technological space.

[3] https://github.com/openflexo-team

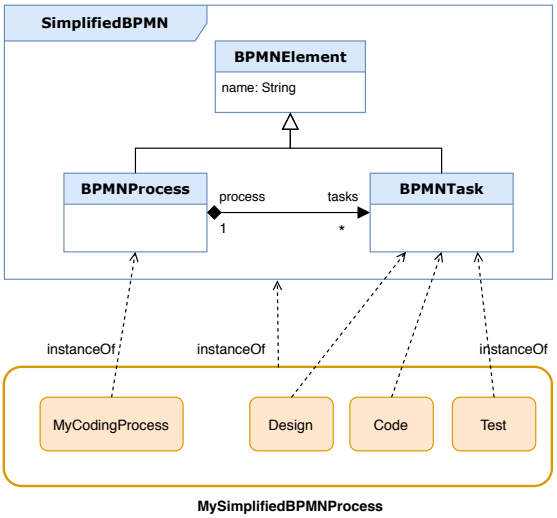

Figure 3: An example of BPMN subset showing both the model and an instance of that model

model federation. This tool offers an FML execution engine with an interactive virtual model design environment.

Finally, tools have their own model. We have taken advantage of Openflexo features to build, in addition to the models required by the challenge, a drawing tool that makes our solution (partially) executable.

3 Analysis

During the requirement analysis of the challenge, the foundation of our reflections and modeling intentions is guided by the general model federation approach.

In a first step, the federation approach is mainly based on modeling relationships between several models, independent of their abstraction levels and the modeling architecture.

During the next step of our approach, we try to take into account the reusability of the relationships by identifying their semantics. The goal is the identification of concepts with their behavior.

The last step is to organize or structure these concepts to improve reusability and extensibility, to create virtual models in FML terminology.

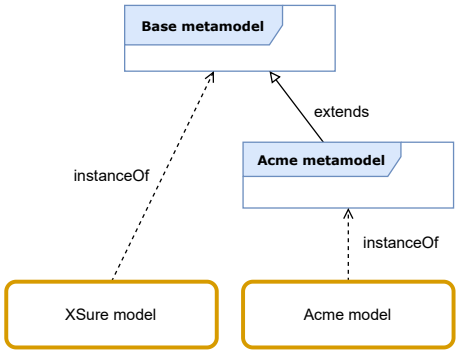

Figure 4: Multilevel architecture of our solution

Like in a lot of modeling approaches, these steps could also be achieved in any order and iteratively. But our goal during the challenge's analysis remains to produce an FML virtual model architecture. As shown in Fig. 4, we defined two virtual models (Base and Acme processes abstractions) instantiated by two virtual model instances (XSure and Acme processes). As the figure shows, the virtual models play the role of metamodels, in the sense of concept definition with a level-agnostic approach. The virtual model instances play the role of models conforming to the metamodel definitions. In the rest of the paper, we use the term metamodel and model to simplify the presentation. The resulting architecture follows the way the challenge is presented and this organization allows for the possibility of flexible extensions.

Our analysis of the use case leads to identify two main modeling axes (as presented in Fig. 5):

- The horizontal axis is characterized as the ontological instantiation axis, in the sense that the domain type definition (i. e. `ProcessType`) is referenced by an instance definition (i. e. `Process`). In our base metamodel, each domain type definition is referenced by its instance definition, as developed in Sect. 4.1.2
- The vertical axis is viewed as the linguistic instantiation. The Metamodel (a virtual model) can be instantiated as a model thanks to the instantiation mechanism of FML. This mechanism is similar to the classical object/instance mechanism of the object languages but without

any constraint on the referenced virtual model, i. e. metamodel. The set of resulting instances comes from several virtual models of any abstraction level as detailed in Sect. 4.2.

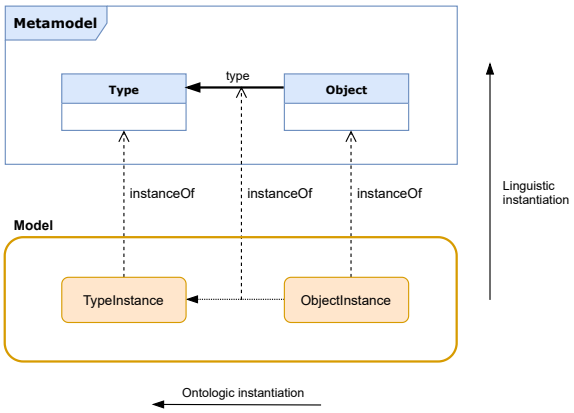

Figure 5: Problem domain linguistic and ontologic instantiation

Based on this approach, we organize our set of models following the architecture of the aforementioned Fig. 4. The resulting multi-level architecture is organized in two virtual models, one defining the core concepts metamodel including the definition of Processes and Tasks, and one, the Acme metamodel, extending the previous model to specialize the Acme definitions. The FML language defines multiple inheritance concept between virtual models as illustrated in Sect. 4.2.

Finally, as explained previously, the defined level for the Acme and the XSure models is the result of the instantiation of virtual models. In our approach, this virtual model instance level cannot be specialized or extended, but all the other virtual models could be extended by any concepts, as a new federated model. Furthermore, the instance level can be instantiated from any virtual model, which represents any metamodel level.

The choices we made are complicated to justify out of context. That is why our choices are presented within the next section model presentation.

4 Model presentation

The presentation of our solution to the Multi-Level Process Challenge follows a systematic methodology where modeling choices are introduced step by step, as the requirements are stated. The satisfaction of requirements is explained throughout this presentation.

The designed multilevel architecture shown in Fig. 4 captures the two use cases described in the Process Challenge. Note that in all following figures, blue is used to represent FML virtual models and concepts (VirtualModel and FlexoConcept in Fig. 2), see Figs. 3–7 and Fig. 9, while brown is used for instances of FlexoConcept, see Figs. 3, 4, 5, 7 and 9.

4.1 Base metamodel for the Process Challenge

In this section, we present the base metamodel presented in Fig. 6 with the *XSure* insurance domain use case, whose partial description was provided in the challenge description. The requirements P1 to P19 of the XSure insurance domain are straightforwardly implemented by instantiating the *XSure model*, as an instance of base metamodel (the left side of Fig. 4).

Fig. 6 represents this base metamodel with a UML-like formalism, well adapted to represent FML concepts and their instances. A *Concept* of FML is represented by a UML class where roles of basic types are attributes. The roles whose types are concepts are represented by an arrow from their concept to their type, with their name on the arrow. The cardinality follows the UML practice. For example, the role `parentActorTypes` of the concept `ActorType` has the type `ActorType` and the * cardinality.

Our proposition relies on ontological instantiation as presented in Fig. 5, with a common root concept `ModelingElement`. Two types of ontological instantiations are needed to meet the requirements of the challenge. The first one is characterized by the fact that instances conform to only one type (`Process` and `Task` do respectively conform to `ProcessType` and `TaskType`). While

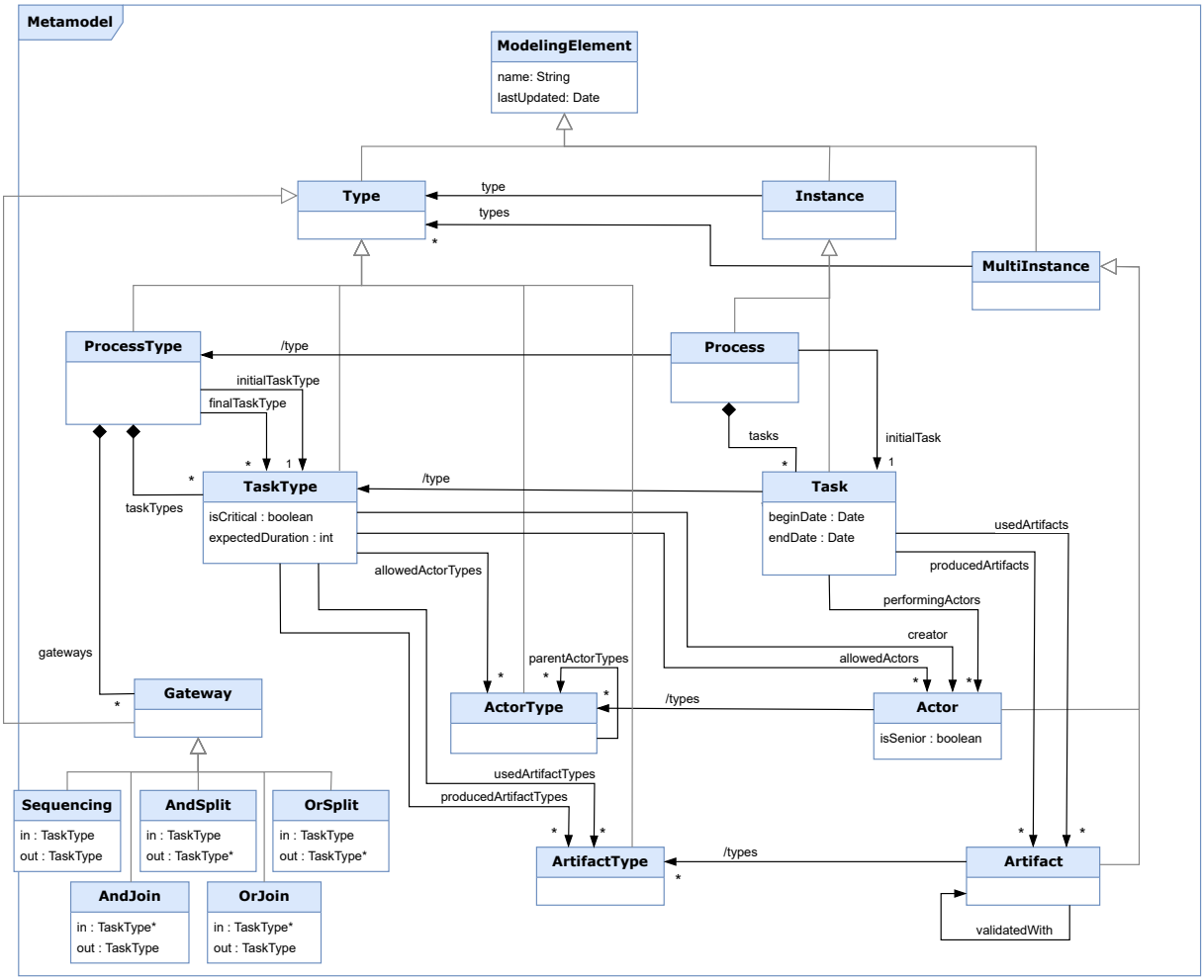

Figure 6: Process management base metamodel (`Gateway`'s associations with `TaskType` are represented with attributes)

the second type of ontological instantiations allows some entities to define their conformity to several types (`Actor` and `Artifact` do respectively conform to several `ActorType` and `ArtifactType`). These instantiations are respectively expressed using the relations `type` and `types` between `Type`, `Instance` and `MultiInstance` concepts. In the diagram, the realization of these relations are indicated by a derived relation, `/type` or `/types`, as for example between `Process` and `ProcessType` or between `Actor` and `ActorType`.

4.1.1 Process type definition

In this subsection, we present the process definition part of the base metamodel, located on the left of Fig. 6.

`ProcessType` is a specialization of the `Type` concept, and references a collection of `TaskType` through the composition relation `taskTypes` with (0..*) cardinality (P1). A `TaskType` is embedded in a `ProcessType` and inherits from its context. To illustrate this in the *XSure* insurance domain use case, the *XSure model* defines the 'Claim Handling', instance of `ProcessType`, and the 'Receive Claim', 'Assess Claim' and 'Pay Premium' instances of `TaskType`.

`ProcessType` also references a collection of gateways, reified with the `Gateway` concept hierarchy. `Gateway` is a specialization of `Type` and is specialized by `Sequencing`, `AndSplit`, `AndJoin`, `OrSplit` and `OrJoin` concepts (P2). Depending on its type and following its underlying operational semantics, a gateway defines one or more inputs and one or more outputs. A `ProcessType` additionally exposes a unique initial `TaskType` and a collection of final `TaskType` with both roles `initialTaskType` (single cardinality) and `finalTaskType` (cardinality 0..*) (P3). `TaskType` exposes a creator role as a reference to an `Actor` concept (P4), which is at the same conceptual level.

The following listing shows an excerpt of the FML code modeling some core concepts of the process modeling base metamodel.

```
model MetaModel {
  concept ModelingElement { ... }
  concept Type extends ModelingElement { ... }
  concept ProcessType extends Type {
    TaskType[0..*] taskTypes;
    TaskType initialTaskType;
    TaskType[0..*] finalTaskTypes;
    Gateway[0..*] gateways;
    concept TaskType extends Type {
      Actor creator;
      Actor[0..*] allowedActors;
      ActorType[0..*] allowedActorTypes;
      ...
    }
    abstract concept Gateway extends Type {
      abstract void execute(Process process);
      ...
    }
    concept Sequencing extends Gateway {
      TaskType in;
      TaskType out;
    }
    // Other core concepts
  }
}
```

The `ActorType` concept inherits from the `Type` concept and the `allowedActorTypes` relation to `ActorType` defined in `TaskType` (with 0..* cardinality) captures P5 requirement. Requirement P6 is symmetrically satisfied with `allowedActors` relation to `Actor` also defined in `TaskType` (with 0..* cardinality). The same modeling pattern applies to `ArtefactType` also extending the concept `Type`, and both relations `usedArtifactTypes` and `producedArtifactTypes` defined in `TaskType` (P7). `TaskType` additionally exposes an `expectedDuration` role (expressed in number of days), satisfying P8.

The `Actor` concept defines a boolean attribute called `isSenior`, while `TaskType` defines an additional `isCritical` boolean attribute, indicating that some instances are flagged as critical and must be performed by senior actors. To fulfill P9 requirement, a supplementary constraint is required for `TaskType` and is captured through the following invariant expressed in the FML language:

```
forEach (actor : allowedActors) {
    assert !isCritical | actor.isSenior
}
```

This invariant should be extended with additional constraints defined in the `Task` concept, which apply to actors assigned to enact tasks.

4.1.2 Process enactment

In this subsection, we present the process enactment part of base metamodel, located right of Fig. 6. All concepts defined in this subsection are either specializations of the `Instance` concept (if they have exactly one type) or the `MultiInstance` concept (when they have several types).

`Process` represents an enacted `ProcessType`, as defined in previous subsection (P10). FML defines behavioral features called behaviors. `ProcessType` defines the behavior `newProcess(String)`, taking the name of the process to enact as argument:

```
public Process newProcess(String name) {
  Process newProcess = new Process(name,this);
  for (taskType : taskTypes) {
    Task newTask = taskType.newTask(newProcess.name
        +"-"+taskType.name),newProcess);
  }
  return newProcess;
}
```

This scheme relies on FML dynamic binding mechanism to delegate to task types the responsibility of instances creation. An instance of `Process` references its unique type `ProcessType` by the specialized `/type` role. Each instance of `TaskType` is ontologically instantiated with a `Task` (P11), using the same factory pattern where `TaskType` has the responsibility to manage the ontological instantiation. A `Task` references its unique `TaskType`, and defines a `beginDate` and an `endDate` basic roles (P12).

A `Task` defines the roles `usedArtifacts`, `producedArtifacts` and `performingActors` (P13). An instance of `Artifact` specializes `MultiInstance` and references a set of `ArtifactType` through the specialized `/types` role (P14 and P16). Likewise, `Actor` specializes `MultiInstance` and references a set of `ActorType` through the specialized `/types` relation (P15).

The semantics is relatively unclear as to the instantiation policy for artifacts. In the requirements, nothing is explicitly stated about the link between the *artifact type* of a *task type* and the types of the artifacts of a corresponding task. Here, we assume that task execution implies that for each used and produced *artifact type* defined in a related *task type*, it exists at least one artifact of the right type. The following excerpt of FML code shows a partial implementation of this. We proceed likewise for produced artifacts.

```
concept Task extends Instance {
  ...
  boolean declaresRequiredUsedArtifacts() {
    for (artifactType : type.usedArtifactType) {
      boolean found = false;
      for (artifact : usedArtifacts) {
        if (artifact.isOfType(artifactType))
          found = true;
      }
      if (!found) return false;
    }
    return true;
  }
  ...
}
```

Authorization for an actor to perform a task (P17) is captured either by the role `allowedActors` or the role `allowedActorTypes` defined in `TaskType`. This mechanism is completed by the behaviors `isAuthorizedActor`, `isValidActor` and `isValidActorType` defined in `TaskType`:

```
concept TaskType extends Type {
 ...
 // Check that an Actor is authorized to perform a
      task, using allowed Actor and ActorTypes
 boolean isAuthorizedActor(Actor actor) {
   for (actType : allowedActorTypes) {
     if (actor.hasActorType(actType))
       return this.isValidActorType(actType);
   }
   for (act : allowedActors) {
     if (actor == act)
       return this.isValidActor(actor);
   }
   return false;
 }
 // Check that an Actor may perform this TaskType
     (override when required)
 boolean isValidActor(Actor actor) {
   return true;
 }
 // Check that an ActorType may perform this
     TaskType (override when required)
 boolean isValidActorType(ActorType actorType) {
   return true;
 }
 ...
}
```

The `Task` concept delegates this authorization to its task type, as shown below.

```
concept Task extends Instance {
  ...
  boolean isAuthorizedActor(Actor actor) {
    return type.isAuthorizedActor(actor);
  }
  ...
}
```

Enforcing those constraints is finally performed by the following invariant of the `Task` concept:

```
forEach (actor : performingActors) {
  assert isAuthorizedActor(actor);
}
```

The default behavior states that all actors and actor types are valid for all task types. This modeling scheme offers many extension points, by the redefinition of some behaviors in the inherited concepts (although none were required in the context of *XSure* use case).

Actor types specialization is captured by the `parentActorTypes` relation defined in `ActorType` (P18). This is completed by both the definition of the `hasActorType(ActorType)` behavior in `Actor` and the recursive behavior `isOrSpecializes(ActorType)` in `ActorType`:

```
concept ActorType extends Type {
  ActorType[0..*] parentActorTypes;
  ...
  boolean isOrSpecializes(ActorType actorType) {
    if (actorType == this)
      return true;
    for (p : parentActorTypes) {
      if (p.isOrSpecializes(actorType)
        return true;
    }
    return false;
  }
  ...
}
concept Actor extends MultiInstance {
  ...
  boolean hasActorType(ActorType actType) {
    for (type : types) {
      if (type.isOrSpecializes(actType))
        return true;
    }
    return false;
  }
  ...
}
```

Each concept inherits from `ModelingElement`, which defines a `lastUpdated` attribute with `Date` type, and thus satisfies P19 requirement.

4.2 The Acme software development process

The challenge describes in a second part a Software engineering process for a fictional Acme company. The Base metamodel described previously is too generic to capture all domain-specific aspects of this use case. We chose to complete the architectural hierarchy with a specific virtual model, specific to the Acme software development process metamodel, shown in Fig. 7. The *Acme* metamodel specializes the Base metamodel, and the *Acme model* is an instance of the *Acme* metamodel. The metamodel inheritance is implemented by virtual model's inheritance. Any instance of the Acme model is either an instance of a concept defined in the Base metamodel, or a concept defined in the specialized *Acme* metamodel.

Fig. 7 highlights all conceptual levels of the architecture. This figure is only partial at instance level and not all instances are represented. It also contains an instance of an enacted *Software Engineering Process* at the bottom. The Acme metamodel specializes some concepts of the Base metamodel, for example, `SETaskType` extends `TaskType` and `SEArtifactType` extends `ArtifactType`. These concepts are further specialized: `CodingTaskType` extends `SETaskType` and `CodeArtifactType` extends `SEArtifactType`. The Acme metamodel also defines a specialized task type `Coding` extending `Task`, and provides the `Developer` concept extending `ActorType`. This metamodel is completed with the `ProgrammingLanguage` concept, defined as an enumeration (`Java`, `C`, `COBOL`). All the instances required to capture the challenge use case are defined in the final Acme model (lower part of Fig. 7). Instances are represented with rounded boxes and linguistic instantiation is represented using dashed connectors.

The *Software Development Process* for Acme company is presented in Fig. 8. It is a screen

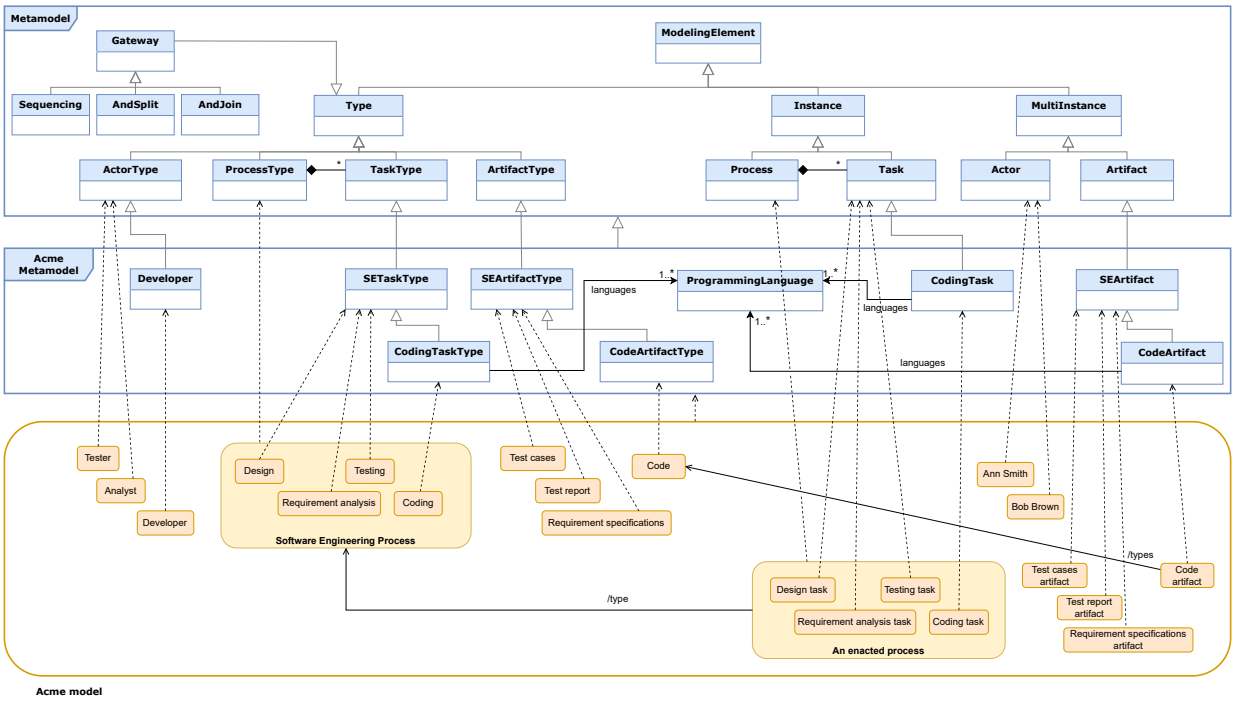

Figure 7: Acme software development process architecture (not all instances are shown for the sake of readability)

capture of the tool developed for the challenge and described in the next section.

'Requirements analysis' is an instance of `SETaskType`, 'Analyst' is an instance of `ActorType` and 'Requirement specifications' is an instance of `SEArtifactType`. The value of the property `producedArtifactTypes` of 'Requirements analysis' is {'Requirement specifications'} and is {'Analyst'} for `allowedActorTypes` (S1). Similarly, 'Test case design' is an instance of `SETaskType`, with a value of `true` for its property `isCritical`, and bound to 'Analyst' by the `allowedActorTypes` relation. 'Test case design' produces 'Test cases' instance of `SETaskType` (S2). The latter is used in 'Test design review' (`producedArtifactTypes` relation) (S1 and S13, satisfied with P9).

Requirement S3 is satisfied by defining 'Coding' as an instance of concept `CodingTaskType` and defining a relation `languages` to `ProgrammingLanguage` with 1..* cardinality. The `newTask(String)` behavior overrides the generic implementation by instantiating a `CodingTask` concept, specializing `Task`, as shown in the following excerpt of FML code:

```
concept CodingTaskType extends SETaskType {
  ProgrammingLanguage[1..*] languages;
  ...
  public CodingTask newTask(String name) {
    return new CodingTask(name,this);
  }
}
```

The `CodeArtifact` concept extends `SEArtifact`, and defines a relation `languages` to `ProgrammingLanguage` with 1..* cardinality (S4).

S5 is more ambiguous as a task type `CodingTask` defines one or more programming languages but produces code which is expressed in one language only. This requirement is captured through the redefinition of the behavior `declaresRequiredProducedArtifacts` where programming language should also match. S6 is guaranteed though the `languages` relation defined in `CodeArtifact` and the following invariant declared in `CodeArtifact`:

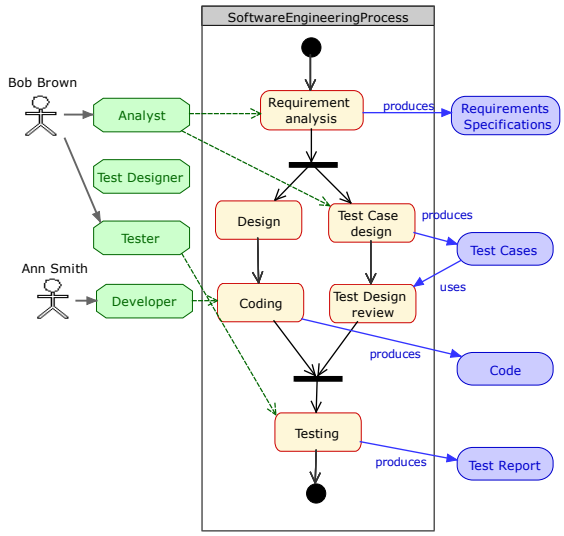

Figure 8: Acme software development process

```
forEach (artifactType : types) {
  assert !(artifactType instanceof CodeArtifactType
      ) | artifactType.doesImplement(languages)
}
```

'Ann Smith' is the only one allowed to perform coding in COBOL (S7). This is implemented through the redefinition of `newTask(String)` in `CodingTaskType`:

```
concept CodingTaskType extends SETaskType {
  ...
  public CodingTask newTask(String name) {
    CodingTask result = new CodingTask(name,this);
    if (languages.contains(ProgrammingLanguage.
        COBOL))
      result.addToPerformingActors(getActor("Ann␣
          Smith"));
    return result;
  }
}
```

The requirement S8 is simply expressed by the definition of the 'Testing' instance of `SETaskType`, 'Tester' instance of `ActorType`, and 'Test report' instance of `ArtifactType`, as shown in Fig. 8.

A critical task must additionally produce artifacts that must be associated with a validation task. This is modeled with a supplementary relation `validatedWith` defined in `Artifact` and referencing the `Artifact` validating it (S9).

All software engineering artifacts defined in the context of Acme Software Engineering Process are instances of `SEArtifact`. This concept defines two attributes: `responsible` (an `Actor` instance), and `versionNumber` (an integer). It thus fulfills S10. 'Bob Brown' is declared as an instance of `Actor`, and references 'Analyst' and 'Tester' (`ActorType` instances). He is also referenced by all instances of `SETaskType` as the creator for related task types (S11). This is done manually by the process designer.

For S12, we assume that all tasks may define an expected duration, which might be checked during process execution. This is modeled by the `expectedDuration` attribute in `TaskType`. Alternatively, this could be modeled through the definition of a `SETesting` concept, as a specialization of `SETaskType`, and the instantiation of 'Testing' as an instance of `SETesting`. The business logic expressed by S12 requirement should then be redefined in `SETesting`.

S13 requirement has been previously partially fulfilled. This must be completed with an association between an artifact produced and the task that validates it. This is modeled by the reference to 'Test Cases' as a produced artifact type in 'Test Case design' and the reference to 'Test Cases' as a used artifact in 'Test Design review', as shown in Fig. 8.

4.3 Openflexo toolset

Our solution is fully implemented within the Openflexo tool. Both use cases have been modeled in the interactive design environment and can be executed by the FML execution engine.

We took advantage of model federation and the availability of diagramming features through the *Diagramming Technology Adapter* to implement two interactive graphical tools built on top of the conceptual levels detailed in previous sections. Fig. 9 shows the overview of the tool architecture and is detailed in the next section. A first tool offers a graphical edition of a Process type. The second one offers an enactment feature (the instantiation of a process from its process type), the ability within an enacted process to assign

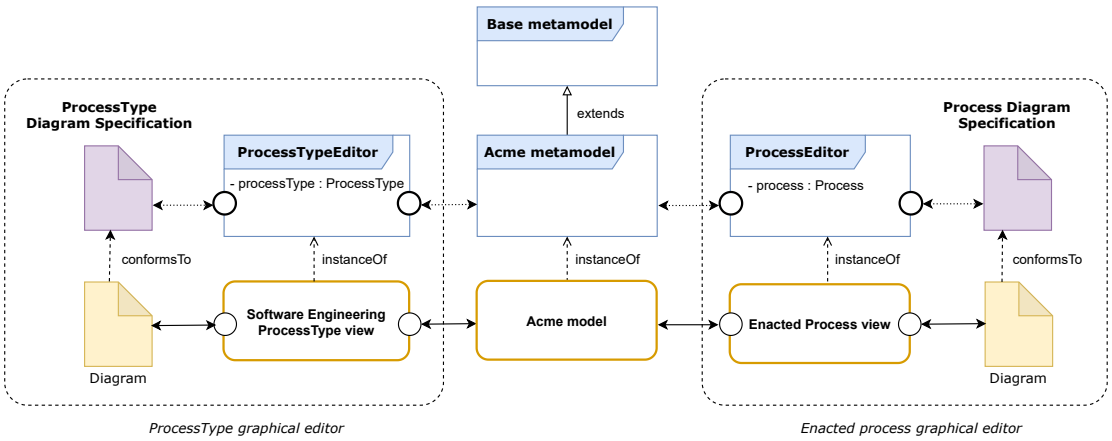

Figure 9: Tool architecture

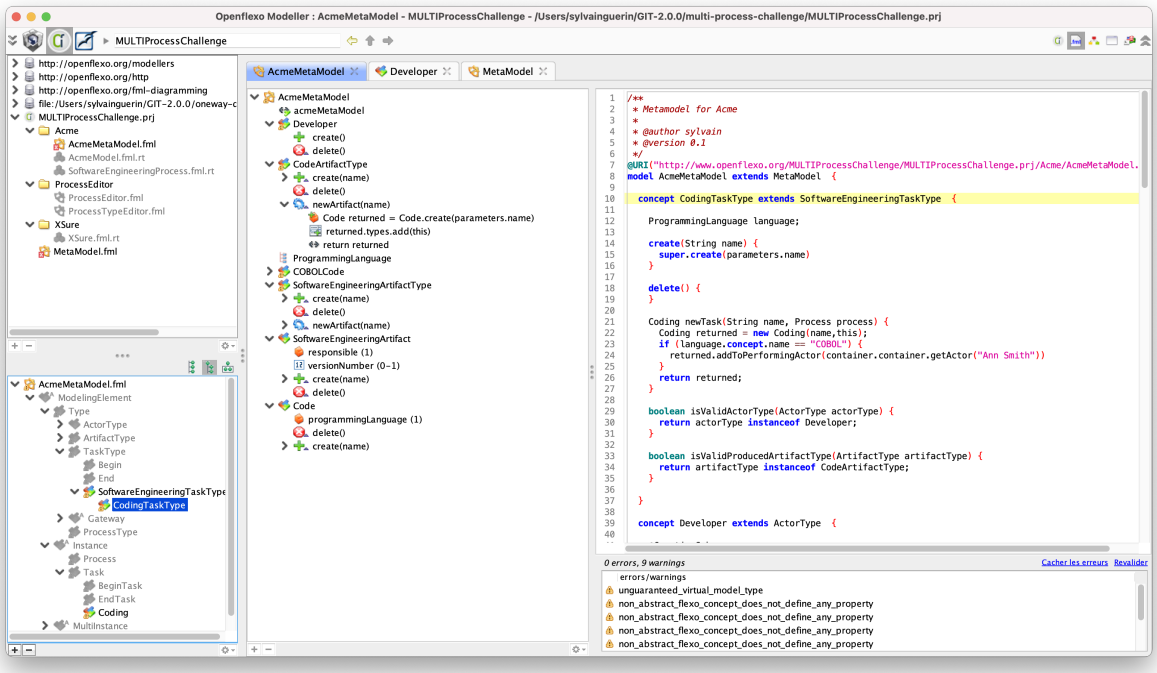

Figure 10: Screenshot of the FML metamodel editor

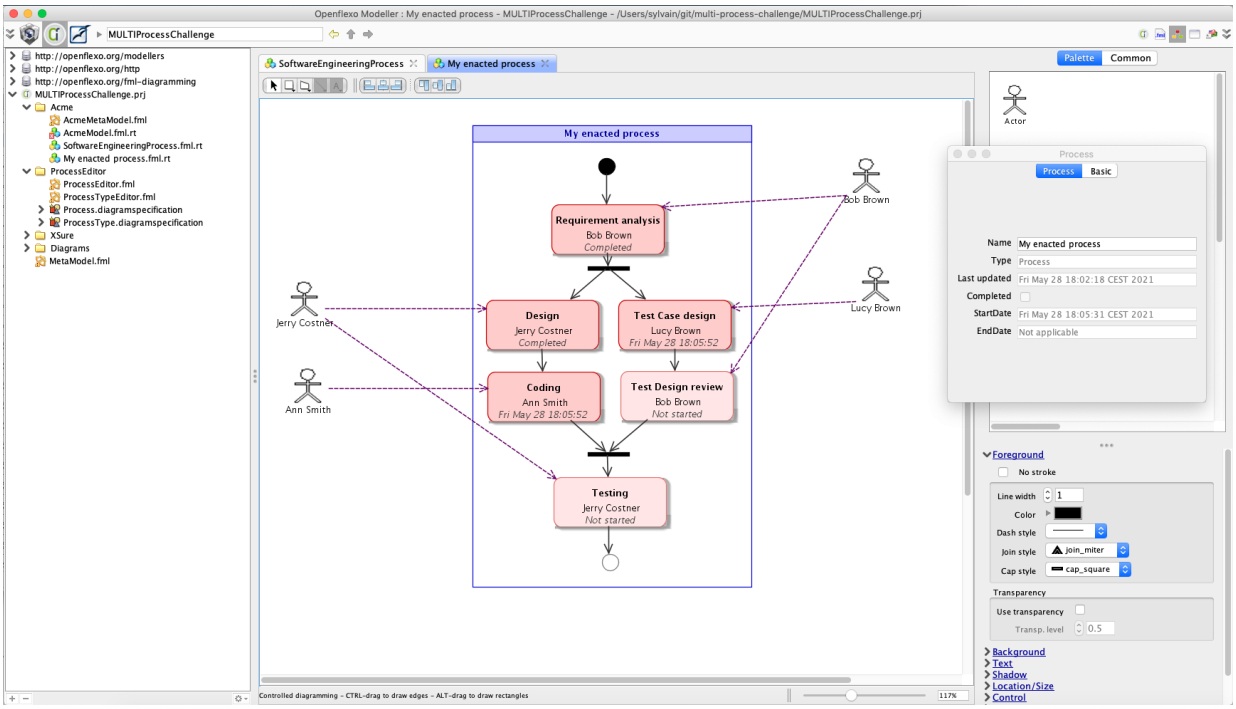

Figure 11: Screenshot of the enacted process graphical editor

tasks to actors, and a graphical visualization of this process execution.

The Base metamodel is completed with some behaviors implementing an execution semantics for the executed processes. All tasks manage a status and are instantiated from `TaskType` for a given enacted process. This status is either `Not startable` (when not assigned to a performing actor or when required input artifacts are not available), `Startable`, `Started`, `Completable` (when all output artifacts are ready) or `Completed`. A `Task` also manages a set of performing actors, a begin and end date, some used and produced artifacts. The `Gateway` business logic has also been implemented through the implementation of an abstract `execute(Process)` behavior. The management of artifacts whose semantics follows rules defined in Sect. 4.1.2 (P13) is also implemented.

Fig. 9 details the architecture of these two tools. The left-hand side of the figure presents the *ProcessType graphical editor*. `ProcessTypeEditor` is modeled as a virtual model declaring two model slots (represented with bold circles). The first model slot references the Acme metamodel and the second model slot references the *ProcessType Diagram Specification*.[4] When executed, this tool manages a graphical view for an instance of the Acme metamodel (the *Acme model*), a specific `ProcessType` instance and a diagram conforming to the *ProcessType Diagram Specification*. This tool allows representing and editing a `ProcessType`. The highly reflective nature of FML and its tools should be noted. When the drag and drop interactor is applied for a new item, the tool allows choosing the concept type to be instantiated. For example, when a `TaskType` is created, the user must choose a concept inheriting from `TaskType` (in the Acme use case it can be `SETaskTask` or `CodingTaskType` or the default value `TaskType`).

The other tool, called *Enacted process graphical editor* and shown on Fig. 11, provides process edition and tasks assignations through a graphical visualization displaying the process being

[4] a diagram "metamodel" which defines and specify structure and graphical representations for edited items.

executed. This tool is represented on the right side of Fig. 9. It follows the same architecture pattern of the *ProcessType graphical editor*, with two model slots referencing both the model and a diagram. Process enactment is operated from a `ProcessType`, and must be defined using a name identifying the newly instantiated process. All tasks are created from their `TaskType` definition, and required assignments apply (if for example 'Coding' `TaskType` defines COBOL as output programming language, the related task is automatically assigned to 'Ann Smith'). Each task has a status as well as a set of performing actors, a begin and end dates, and used and produced artifacts.

A demonstrative video for the developed tool is available on our website,[5] as well as download and installation instructions.

5 Satisfaction of requirements

In the previous Sect. 4, we demonstrated that our solution satisfies all the challenge requirements. In this section, we summarize the different techniques used to satisfy them, without repeating all the justifications. We classify these techniques into three categories:

- *syntactic conformance*: the requirements are structurally satisfied when the model conforms to its metamodel;
- *constraints checking*: the requirements are fulfilled when all instances satisfy both syntactic conformance and the evaluation of invariants expressed for related concepts;
- *process tools*: the requirements are enforced by the implementation of the process tools illustrated by Fig. 9. It can be done by using behaviors defined in the models or by using the process tools manually.

Table 1 summarizes the requirement coverage for the base metamodel illustrated by the XSure insurance use case, while Table 2 shows requirements satisfaction for the Acme software engineering process. Syntactic conformance is more precisely classified into six subcategories:

- *Conceptualization* (*Conc.*): a concept is created to fulfill or meet the requirement (for example for P1: *process type* is conceptualized in a concept `ProcessType`).
- *Specialization* (*Spec.*): a concept is specialized in the sense of object-oriented modeling (for example for P2: `Gateway` defines an abstract behavior `execute()` and `Sequencing`, `AndSplit`, `AndJoin`, `OrSplit` and `OrJoin` are defined as inherited concepts implementing specific business logic).
- *Composition* (*Comp.*): a new relation between existing concepts or specific attributes in existing concepts are added (for example for P3: a *process type* has one *initial task type*).
- *Linguistic instantiation* (*L.Inst.*): a requirement is satisfied by the instantiation of one or more concepts (for example, for S1: 'Requirement analysis' is defined as an instance of `SETaskType`).
- *Ontological instantiation* (*O.Inst.*): fulfillment of a requirement is obtained by the relation between an instance and its type definition instance (for example, the notion of *developer* of S3 is implemented by both a concept in the *Acme metamodel* and an instance in the *Acme model*, see bottom left of Fig. 7).
- *Behavioral modeling* (*B.Mod.*): a requirement is satisfied by the definition of one or more behaviors, which may be combined with constraints checking and associated tools (for example P17).

6 Assessment of the modeling solution

In this section, we discuss our multi-level model solution with regard to the required aspects mentioned in the challenge.

[5] https://research.openflexo.org/MLMChallenge.html

	Syntactic conformance						Constraints checking	Process tools
	Conc.	Spec.	Comp.	L.Inst.	O.Inst.	B.Mod.		
P1	✓		✓					
P2	✓	✓	✓					
P3			✓					
P4	✓				✓			
P5	✓		✓					
P6	✓		✓					
P7	✓		✓					
P8			✓					
P9			✓				✓	
P10	✓				✓			
P11	✓		✓		✓			
P12			✓					
P13	✓		✓		✓			
P14	✓				✓			
P15	✓				✓			
P16	✓				✓			
P17	✓					✓	✓	✓
P18	✓					✓		
P19		✓	✓			✓		

Table 1: Requirements satisfaction for the base metamodel

	Syntactic conformance						Constraints checking	Process tools
	Conc.	Spec.	Comp.	L.Inst	O.Inst	B.Mod		
S1				✓				
S2				✓				
S3	✓	✓	✓	✓				✓
S4	✓	✓	✓	✓				
S5		✓		✓		✓		
S6		✓		✓		✓	✓	
S7				✓		✓		
S8				✓				
S9						✓	✓	
S10	✓	✓	✓					
S11				✓				✓
S12				✓				✓
S13				✓				

Table 2: Requirements satisfaction for the Acme software engineering process

6.1 Basic modeling constructs

Our solution uses the basic modeling constructs depicted by the FML core metamodel (Fig. 2) and described in Sect. 2. Models are *virtual models*. They are composed of *concepts*, being themselves concepts. Concepts may have *roles* and *behaviors* (actions one can perform).

In order to provide the graphical tools, our solution also uses *model slots* to connect some concept instances to external elements (in our case: instances of diagram from our diagramming Technology Adapter). The use of slots has been described in Sect. 4 and is illustrated by bold circles in Fig. 9.

6.2 Levels

The Openflexo approach is level agnostic: "levels" have no specific nature and there are no numbered levels. In our solution, concepts of a given level are grouped into a virtual model. Inheritance and instantiation allow the establishment of relationships between concepts from different levels. Fig. 4 shows the multi-level architecture of our solution.

6.3 Number of levels

Due to the fact that our approach is level-agnostic, our solution could have more or fewer levels depending on the variations of the use case. However, the number of levels is related to the problem. For example, our solution uses 3 levels for this multi-level process challenge.

6.4 Cross-level relationships

Thanks to the level-agnosticism nature of the Openflexo approach, cross-level relationships are not an issue. Model elements of different levels can be linked to each other in a transparent way, using inheritance or instantiation. For instance, in Fig. 7, the `Developer` concept from `Acme` metamodel specializes the `ActorType` of the base metamodel. The `Software Engineering Process` of the `Acme` model is an instance of `ProcessType` in the metamodel.

6.5 Cross-level constraints

As for cross-level relationships, cross-level constraints do not have a specific nature. They are like any constraint that a user can define by adding a behavior to a model element. Therefore, if a constraint is mandatory when establishing a relationship between elements from different levels, the user has to create it manually. This can be a limit of our approach: the cost of the flexibility of our toolset is a limited number of automated behaviors.

6.6 Integrity mechanism

Behaviors are continuously checked, ensuring the integrity of the models when changes occur. However, most of these behaviors have to be written by the users. Thus, the integrity of the contents relies on the users when they build the metamodels and the models. Sect. 4 describes how the requirements are captured and how behaviors are implemented with FML. For instance, `isAuthorizedActor` behavior is needed to enforce P17 requirement, leading to writing FML code in `TaskType` and `Task`. Note that in our approach, metamodels are built in an ad hoc way along with the models (co-construction), therefore the user also validates the integrity continuously.

6.7 Deep characterization

Due to the nature of our approach and its level-agnosticism, *deep characterization* does not apply to our solution. Such a mechanism could probably be encoded by specific concepts and constraints (behaviors). However, it would not be a generic mechanism, making it difficult to reuse for another problem.

6.8 Generality

Due to the nature of model federation, the models are highly reusable. Although we build ad hoc metamodels in our approach, our solution separates the domain-specific elements from general purpose ones, making possible to reuse the general concepts to other domains.

6.9 Extensibility

A strength of the model federation approach we have adopted resides in its flexibility. As we build metamodels and models together instead of fitting into a metamodel, our solution is more flexible. Therefore one can extend a solution easily. The challenge itself can be seen as a validation of the extensibility of our solution: it was decomposed into two steps (the Xsure process, then the ACME process) which can be seen as a simulation of the evolution of the specification. We observed that our approach has been resilient to change.

Due to the level-agnostic aspect of our approach, we could insert a new level, for example. It would consist in creating new concepts we would link to the other ones. Then, we would add any necessary constraints on those concepts and on the relationships linking them to the other existing concepts. This type of change can be done without much effort. On the other hand, making changes to the existing models could be difficult. For example, given a requirement, an instance cannot be changed into a concept easily. This case could occur if the challenge had refined the specification by requiring specific design tasks. In this context, we would have to transform the 'Design Task' instance into a concept whose instances could have been "using agile methodology".

6.10 Tools

We use the Openflexo infrastructure to build models and metamodels together, and to provide dedicated tools to edit a Process type, to enact a process and to execute it. Being able to provide a solution and its associated tools quickly and easily is a noticeable feature. We could also have taken advantage of the model federation to connect our solution to other tools dedicated to process edition and to process execution (e. g. a BPMN engine). However, we already had all the necessary components to provide the aforementioned graphical tools.

6.11 Model verification

Our toolset includes mechanisms to verify the models. First, syntactical consistency is realized by cardinality checks and by typing. Second, the constraints the users have written are continuously checked during model edition and enactment. Thus, a part of the verification relies on the fact that users add behaviors when they build the metamodels and the models. For instance, S6 requirement is constrained by an invariant implemented in `CodeArtifact`. It ensures that `artifactType` has the right type and that it implements the `languages` relation.

7 Related work

A plethora of multi-level modeling approaches and tools[6] with different foundations have appeared in recent years (Somogyi et al. 2021). Comparing them lies out of the scope of this paper. In this sense, we limit the following discussion to the presentation and comparison of previous solutions to the MULTI Process Challenge (Almeida et al. 2019).

A first description of process modeling as a multi-level modeling problem was proposed by Lara and Guerra (2018) in the context of a catalogue of refactoring for multi-level models (in a simpler form with fewer constraints and requirements w.r.t. the challenge version). A solution is provided with MetaDepth (De Lara and Guerra 2010), which supports modeling with any number of levels, dual ontological/linguistic typing and deep characterization through potency (Atkinson and Kühne 2001). For their solution to the multi-level process problem they use three levels. The first level describes generic processes, the second level software engineering processes, and the third level, software engineering enacted processes. Additionally, the authors use *linguistic extensions* (De Lara and Guerra 2010), a mechanism to linguistically extend ontological instance models at any level, in order to introduce artifact types and tasks duration in the second level. Our solution is similar to theirs w.r.t. the grouping and organization of concepts (e. g., a model for general process, concepts, an extension for

[6] https://homepages.ecs.vuw.ac.nz/Groups/MultiLevelModeling/MultiTools

software engineering process) but it replaces the use of clabjects and potency with the use of the type-object pattern (Johnson and Woolf 1997). The so-called *linguistic extensions* are natively supported in Openflexo/FML.

Rodríguez and Macías (2019) use MultEcore (Macías et al. 2016) to provide a solution to the challenge. MultEcore uses (un)pluggable linguistic levels (e. g., there is not a *fixed* core metamodel defining the concept of clabject), an extension of the two-level cascading technique (Atkinson and Kühne 2005) (this is achieved with model transformations to transform instance models into instantiable models) and potency. Regarding the challenge, their solution uses 4 levels. The first level represents generic processes and constitutes the root level for two ontological hierarchies, one for the software engineering domain and the other for the insurance domain. Each hierarchy contains three other ontological levels. The second level refines generic process to software engineering and insurance processes. The third level refines the aforementioned models to adapt them to the ACME and Xsure processes. Finally, process instances lie in level 4. Additionally, the authors use an orthogonal linguistic hierarchy to support alternative names for every model element. With respect to our solution, the authors use more models, as we require less refinement steps. However, accidental complexity is introduced by our approach as we need additional constructs and constraints for the typing relations which are otherwise built-in in MultEcore.

More similar to us, Jeusfeld (2019) uses DeepTelos (Jeusfeld and Neumayr 2016), an extension of the Telos language (Mylopoulos et al. 1990) in order to solve the process challenge. As with Openflexo/FML, they do not use explicit (numbered) levels nor potency. However, unlike Openflexo, DeepTelos integrates a multi-level modeling specific construct similar to the PowerType (Atkinson and Kühne 2001) pattern they call *most general instances* which the authors use to simulate potency. Models in DeepTelos are organized in (tree-like) module hierarchies, where submodules can see all the concepts defined in parent modules. The authors use this module system in order to *organize* their solution for the Multi-level process challenge. Concretely they created a hierarchy of modules such as the top module contains base concepts and formulas required for multi-level modeling (e. g., the support for *most general instances*), and a sub-module contains process definitions fulfilling requirements P1 to P19. This sub-module contains in turn a sub-module representing the coding process type whereas concrete coding processes are represented in subsequent sub-modules. There are two main differences between the DeepTelos and Openflexo/FML: 1) we use the type-object pattern instead of the powertype pattern; 2) DeepTelos reifies their support for multi-level modeling while our remains ad hoc.

Finally, Somogyi et al. (2019) use the Dynamic Multi-Layer Algebra (DMLA) (Urbán et al. 2017) in order to solve the process challenge. DMLA is a level-blind modeling framework which is fully customizable (e. g. different types of instantiation may be implemented) and includes support for deep characterization (instantiation may refer to any other element disregarding hierarchy levels and a sort of potency in the form of fluid metamodeling at the entity level exists). Their proposed solution separates the process challenge in two separate domains, namely, the task definition domain and the process definition domain. Wrappers are used in order to reuse tasks in processes. Their solution does not use inheritance, as it is not supported by the framework. It does not use explicitly separated models either.

8 Conclusions

We fulfill all the challenge requirements and we propose a tool that demonstrates the usability of our solution. We developed an ad hoc solution, including models and metamodels, thanks to the flexibility provided by the metametamodeling infrastructure offered by Openflexo. This infrastructure helped us to overcome the limitations of the strict modeling paradigm mentioned

in Sect. 1, this is, the lack of support for different forms of classifications and for the duality type-object. In that sense, our solution seamlessly integrates the type-object pattern which allows us to dynamically create different *instances* of model elements (e. g., *TaskType*, among others) and use them for typing other model elements. This, together with the FML core support for (level agnostic) classical linguistic instantiation, permitted us to merge linguistic and ontological instantiation in order to provide a satisfactory solution to the Multi-Level Process challenge.

The features of FML enable a great modeling flexibility which, coupled with an appropriate methodology, allow us to achieve the multi-level capabilities without dedicated tool, in order to solve problems which require it. We use virtual models to "implement" levels on demand.

As a future work, we envision to explore a number of alternative solutions. Firstly, instead of two levels defined by Type and Instance concepts, we could have a single concept mixing a type part and an instance part. It would be equivalent to implement a form of Clabject, a more flexible approach concerning level separation. This would probably ease adaptation if the problem specifications evolve. Secondly, another possible approach is the use of the free modeling tool proposed by Openflexo. The solution would be developed from examples (the Acme and XSure processes, for instance) from which models would be identified.

Finally, the realization of this challenge highlighted the interest of integrating specific behaviors for the management of multi-level concepts in the Openflexo infrastructure.

References

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

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

Atkinson C., Gerbig R. (2016) Flexible deep modeling with melanee. In: Modellierung 2016 – Workshopband, pp. 117–121

Atkinson C., Kühne T. (2001) The essence of multilevel metamodeling. In: «UML»2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. Springer, pp. 19–33

Atkinson C., Kühne T. (2005) Concepts for comparing modeling tool architectures. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 398–413

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

Golra F. R., Beugnard A., Dagnat F., Guerin S., Guychard C. (2016a) Addressing modularity for heterogeneous multi-model systems using model federation. In: Companion Proceedings of the 15th International Conference on Modularity, pp. 206–211

Golra F. R., Beugnard A., Dagnat F., Guerin S., Guychard C. (2016b) Using Free Modeling As an Agile Method for Developing Domain Specific Modeling Languages. In: Proc. of the ACM/IEEE 19th International Conf. on Model Driven Engineering Languages and Systems. MODELS '16. Saint-Malo, France, pp. 24–34

Golra F. R., Dagnat F. (2011) The lazy initialization multilayered modeling framework. In: Taylor R. N., Gall H. C., Medvidovic N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011. ACM, pp. 924–927

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

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level modeling with most general instances. In: International Conference on Conceptual Modeling. Springer, pp. 198–211

Johnson R., Woolf B. (1997) Type Object In: Pattern Languages of Program Design 3 Martin R., Riehle D., Buschmann F. (eds.), 1st ed. Software Pattern Series Addison-Wesley Longman Publishing Co., Inc., USA chap. 4, pp. 47–65

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

Macías F., Rutle A., Stolz V. (2016) MultEcore: Combining the best of fixed-level and multilevel metamodelling. In: Proceedings of the 3rd International Workshop on Multi-Level Modelling. CEUR Workshop Proceedings Vol. 1722. CEUR-WS.org, pp. 66–75 http://ceur-ws.org/Vol-1722/

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

Object Management Group (2016) Semantic Modeling for Information Federation (SIMF). https://www.omg.org/cgi-bin/doc.cgi?ad/2011-12-10. Last Access: February 23^{rd}, 2022

OMG (2013) OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. http://www.omg.org/spec/MOF/2.4.1. Last Access: February 23^{rd}, 2022

Openflexo (2019) Openflexo Project. https://www.openflexo.org/. Last Access: February 23^{rd}, 2022

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

Somogyi F. A., Mezei G., Theisz Z., Bácsi S., Palatinszky D. (2021) Playground for multi-level modeling constructs. In: Software and Systems Modeling

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

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

Urbán D., Mezei G., Theisz Z. (2017) Formalism for Static Aspects of Dynamic Metamodeling. In: Periodica Polytechnica Electrical Engineering and Computer Science 61(1), pp. 34–47