Multi-level modeling with Openflexo/FML- A contribution to the MULTI 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* MULTI 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 Flexo Modeling Language (FML) that serves to describe them as the main building block in order to solve the process modeling challenge whereas the federation aspect is used as a means to provide editing tooling for the resulting process language. Our solution is fully implemented with the Openflexo framework. It fulfills all the challenge requirements.*

Keywords. Multilevel Modeling Challenge • Federation Modeling Language • Openflexo

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 sub domains. 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 linguistic); 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 artefacts.

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 well as more flexible relations between them. This paradigm is gaining traction lately within the modeling community as evidenced by the con-

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

tribution of many new multi-level modeling approaches and tools such as Melanee (Atkinson and Gerbig 2016), MetaDepth (De Lara and Guerra 2010), MultEcore (Macias Gomez de Villar 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. This paper is a response to the latest multi-level modeling challenge 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. We provide here 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 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 Flexo Modeling Language (FML), a language to create, link and manage virtual models, in order to solve the MULTI process challenge while the federation aspect is used solely so as to provide tooling for the resulting process language. Our solution, which is fully implemented and executable, meets all the requirements.

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

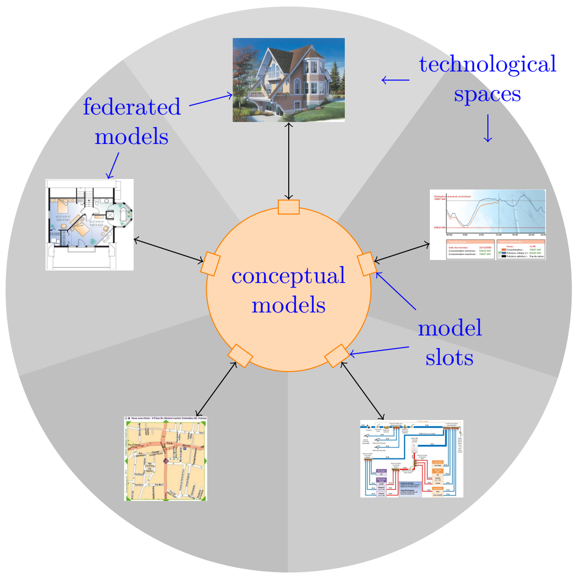


Figure 1: The model federation approach

2 Technology

To meet the MULTI process challenge, we have decided to use the general approach of model federation (Golra et al. 2016a). *Model federation* is a way to assemble models using some kind of low-coupling links. It has been studied to answer the OMG' 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 – freeing oneself from the bonds of model/metamodel conformity – that is presented in (Golra et al. 2016b). Another notable feature of this approach is the strong decoupling among tools that remain usable after federations are made.

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

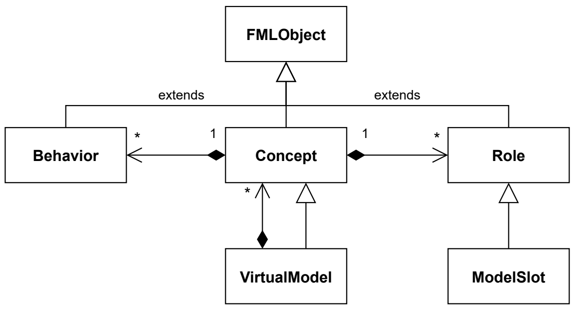


Figure 2: The FML core metamodel

This framework relies on the architecture of Figure 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 Flexo Modeling Language (FML). Each federated model is an autonomous element that may evolve with its own tooling. 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 aspect is on the tooling made for the process language.

FML simplified metamodel is provided in Figure 2. It is designed to define models as virtual models. A virtual model is composed of a set of *concepts*, 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 *roles* and *behaviors*. A parallel to object-oriented approach can be useful to understand FML[1] . A concept corresponds to a class, its roles to the attributes of the class and its behaviors to the methods of the class. These roles have types defining the kind of value the role will point at runtime. Whenever a type external to the federation space is used, one needs to use a *model slot*. A model slot is a mediation entity in charge of giving access to external elements using a *technology adapter*[2] .

FML is designed to define not only the structure of virtual models but also to define the collection of actions an engineer can perform on them. These actions are called *behaviors* and can either be called (like usual methods) or triggered by events. The reactive behaviors are mainly useful when a federated model evolution needs to trigger a computation. We do not exploit this possibility in the challenge.

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.

The tool support for model federation framework, Openflexo[3] , is developed as an open source initiative. This tool offers a 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 parallel with 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, independently of their abstraction level and the model architecture.

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

[1] Some aspects of FML do not exist in object oriented approach.

[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

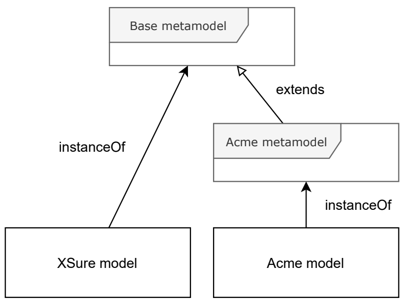


Figure 3: Multilevel architecture of our solution

and extensibility, to create virtual models in FML terminology.

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 a FML virtual model architecture for our federation. As shown in the figure 3, 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 axis (as presented in the figure 4):

- 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 the section 4.1.2
- The vertical axis is viewed as the linguistic instantiation, relative to the use of the FML language. The virtual model, defined as the core definition metamodel, generates a model founded on the instantiation mechanism of the FML Language. 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 the section. 4.2

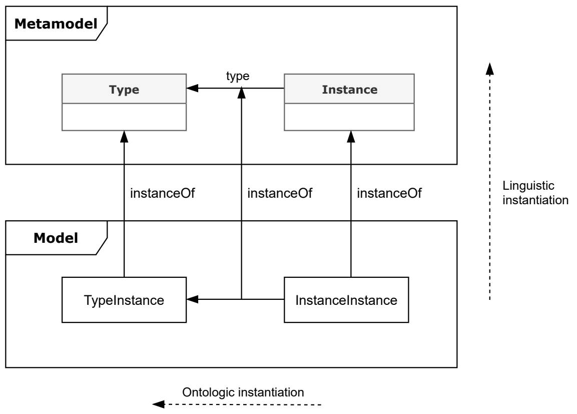


Figure 4: Linguistic and ontologic instantiation

Based on this approach, we organize our set of models following the architecture of the aforementioned Figure 3. The resulting multi-level architecture is organized in two virtual models, one for the core concepts metamodel containing the definition of Processes and Tasks, and one, the Acme metamodel, extending the previous model to integrate the Acme definitions. The FML language defines multiple inheritance concept between virtual models as illustrated in the section 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 can't be specialized or extended but all the other virtual models could be extended by any concepts, as a new federated model. Also, 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 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 Figure 3 captures the two use cases described in the Process Challenge.

4.1 Base metamodel for the Process Challenge

In this section, we present the base metamodel presented in Figure 5 with the *XSure* insurance domain use case, whose partial description was provided in the challenge description. The requirements **P1** to **P19** of XSure insurance domain are straightforwardly implemented by instantiating the *XSure model*, as an instance of base metamodel (the left side of figure 3).

Figure 5 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 a combination of a name in their containing concept and an arrow from their name to their type. 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 figure 4, with a common root concept ModelingElement. Two types of ontological instantiation are needed to meet the requirements of the challenge. One provides that some instances conform to only one type (Process and Task do respectively conform to ProcessType and TaskType). While some entities 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

We first present the process definition part of the base metamodel, located on the left of Figure 5.

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 *XSure* insurance domain use case, *XSure model* defines 'Claim Handling', instance of ProcessType, and '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;
```

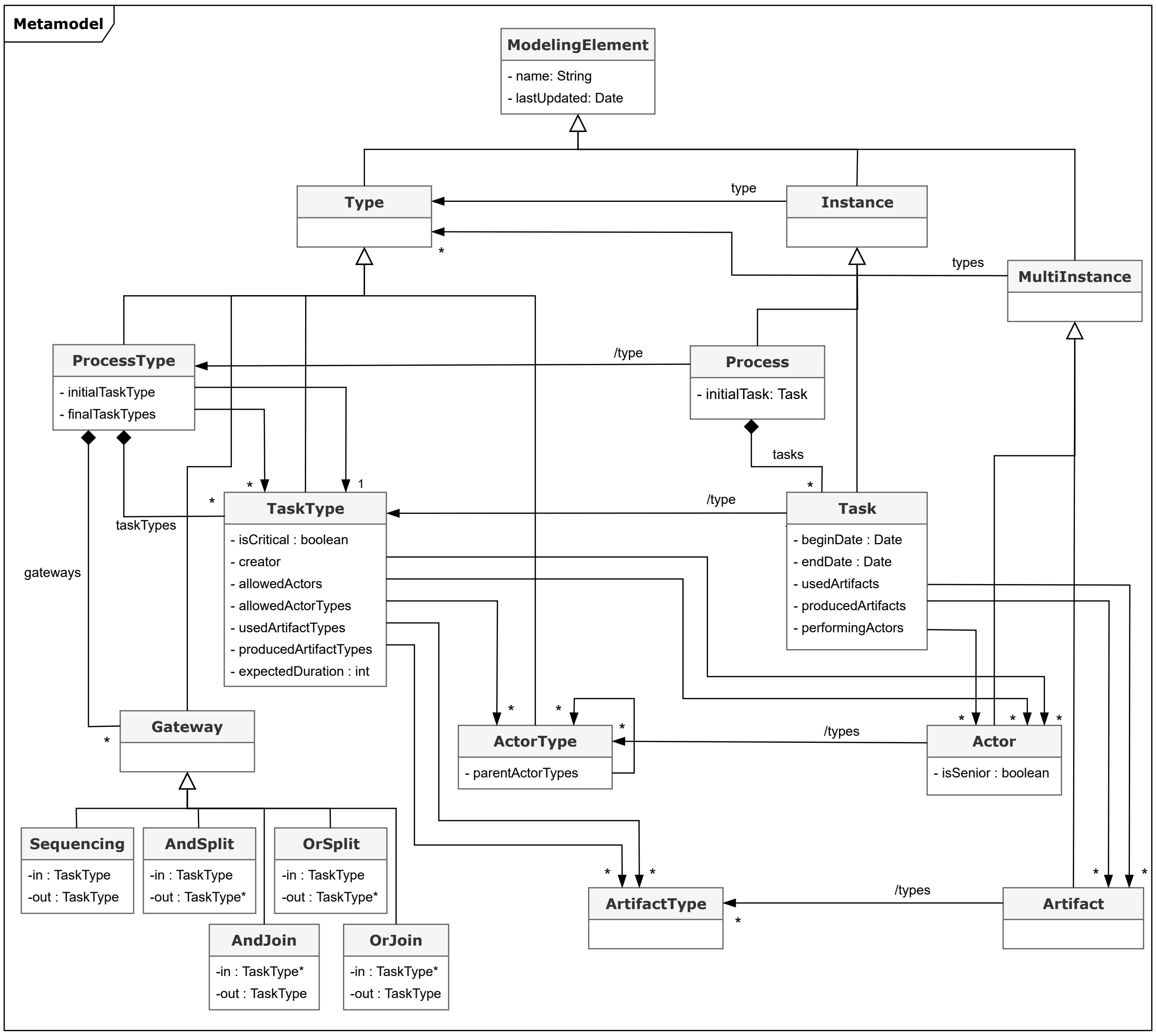


Figure 5: Process management base metamodel

```
    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 is a sub-concept of Type 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 defined as a sub-concept of 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 completed with additional constraints defined in the Task concept, which apply to performing actors assigned to enact tasks.

4.1.2 Process enactment

We now present the process enactment part of base metamodel, located right of figure 5. All concepts defined in this subsection are either specialization 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 behavior. 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 pattern where TaskType has the responsibility to manage the ontological instantiation. A Task references its unique TaskType, and defines a begin date and an end date basic role (**P12**).

The same pattern applies for the used and produced artifacts by roles usedArtifacts, producedArtifacts and performingActors defined in Task (**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 unclear relatively 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. The same pattern applies to 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 in the following FML code:

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

Enforcing those constraints is finally performed by the definition of this invariant in 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 choose to complete the architectural hierarchy with a specific virtual model, specific to the Acme software development process metamodel, shown in figure 6. The *Acme* metamodel specializes the Base metamodel, and *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.

The Figure 6 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 enacted a *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 the figure). Instances are represented with rounded boxes and linguistic instantiation is represented using dashed connectors.

The *Software Development Process* for Acme company is presented in the figure 7. It is a screen 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**).

The capture of **S3** requirement is performed while 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 `language` 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.

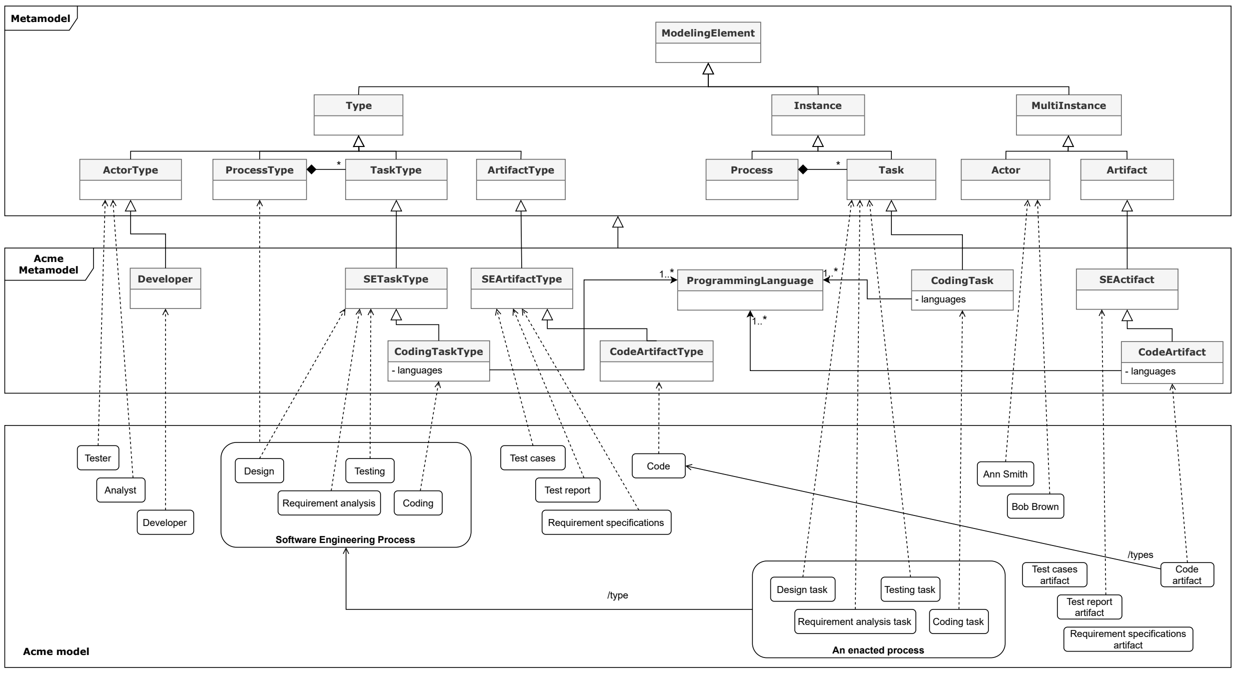


Figure 6: Acme software development process architecture (not all instances are shown for readability reasons)

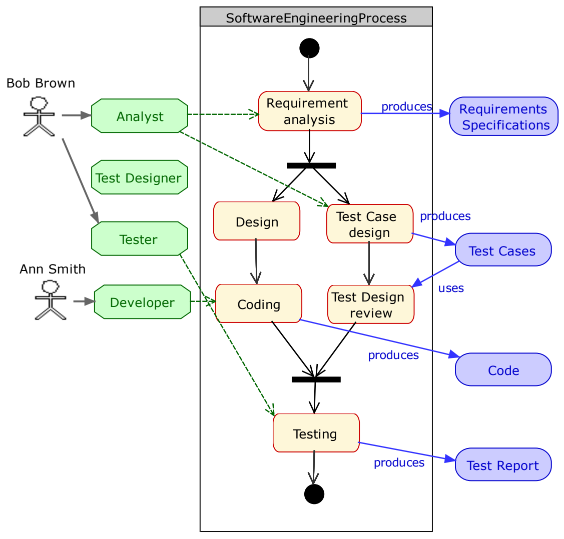


Figure 7: Acme software development process

S6 is guaranteed though the `language` relation defined in `CodeArtifact` and the following invariant declared in `CodeArtifact`:

```
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`.

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 `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**).

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 figure 7.

4.3 Openflexo tooling

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 diagraming features though the *Diagramming Technology Adapter* to implement two interactive graphical tools built on top of the conceptual levels detailed in previous sections. 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 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 – instantiated from `TaskType` for a given enacted process – manage a status. 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 section 4.1.2 (**P13**) is also implemented.

Figure 8 shows the architecture of these two tools. The left 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 an instance of Acme metamodel, while the second model slot references an instance of diagram, conforms to the *ProcessType diagram specification*[4] . When executed, this tool manages a graphical view for an instance of Acme metamodel (the *Acme model*) and a specific `ProcessType` instance. This tool allows representing and editing a `ProcessType`. The highly reflective nature of FML and its tooling 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 sub-concept of `TaskType` – in Acme use case it can be `SETaskTask` or `CodingTaskType` or default value `TaskType`).

The other tool, called *Enacted process graphical editor* and shown on figure 9, provides process edition and tasks assignations through a graphical

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

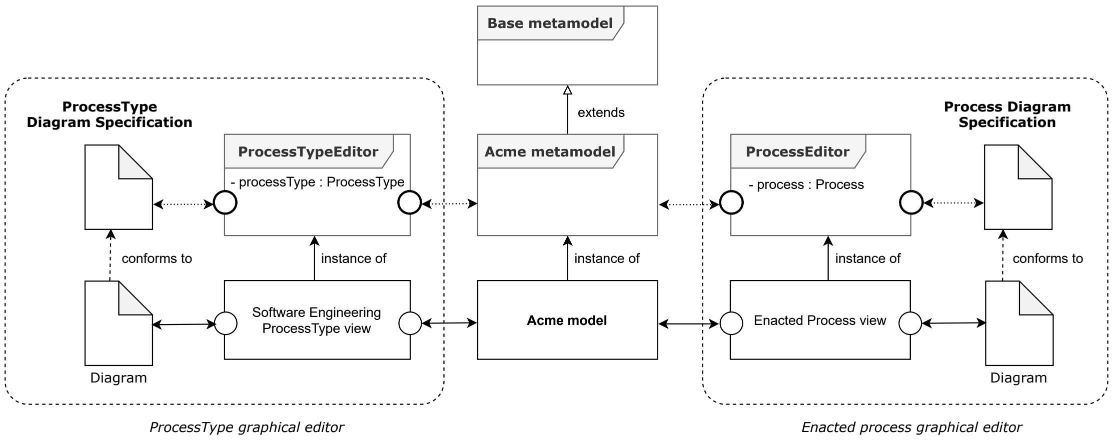
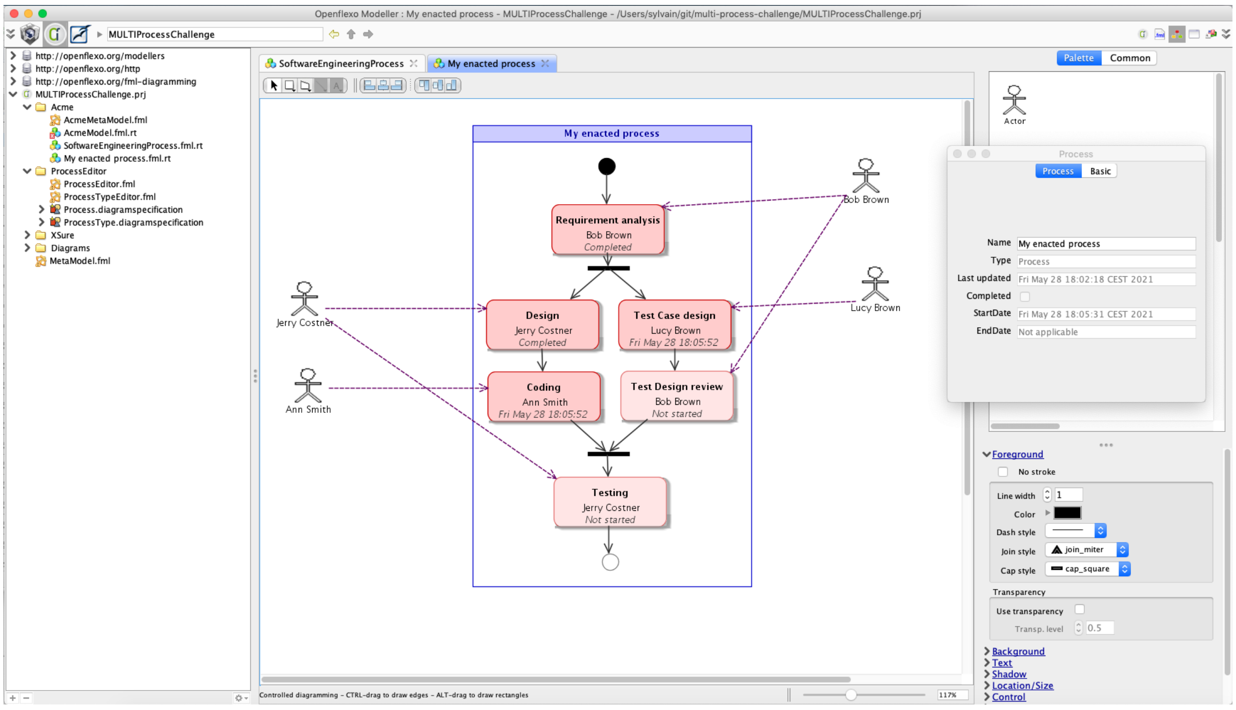


Figure 8: Tooling architecture


Openflexo Modeller : My enacted process - MULTIProcessChallenge - /Users/sylvain/git/multi-process-challenge/MULTIProcessChallenge.prj

MULTIProcessChallenge

- http://openflexo.org/modellers
- http://openflexo.org/http
- http://openflexo.org/fml-diagramming
- MULTIProcessChallenge.prj
  - Acme
    - AcmeMetaModel.fml
    - AcmeModel.fml.rt
    - SoftwareEngineeringProcess.fml.rt
    - My enacted process.fml.rt
  - ProcessEditor
    - ProcessEditor.fml
    - ProcessTypeEditor.fml
    - Process.diagramspecification
    - ProcessType.diagramspecification
  - XSure
  - Diagrams
  - MetaModel.fml

Name	My enacted process
Type	Process
Last updated	Fri May 28 18:02:18 CEST 2021
Completed	☐
StartDate	Fri May 28 18:05:31 CEST 2021
EndDate	Not applicable

Foreground

☐ No stroke

Line width	1
Color	■
Dash style	
Join style	join_miter
Cap style	cap_square

Transparency

Use transparency	☐
Transp. level	0.5

Background
Text
Shadow
Location/Size
Control

Controlled diagramming – CTRL-drag to draw edges – ALT-drag to draw rectangles

117%

Figure 9: Screenshot of the enacted process graphical editor

visualization displaying the process being executed. This tool is represented on the right side of the figure 8. 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 date, and used and produced artifacts.

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

5 Satisfaction of requirements

In the previous section 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,
- *tooling*: the requirements are enforced by the implementation of the associated tooling.

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

- *Conceptualization* (*Conc.*): a concept carries the semantics exposed by 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.*): some concepts are in relation, or specific attributes were added to some concepts (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 for **S3** : *developer* has the dual concept nature in *Acme metamodel* and instance nature in *Acme model*).
- *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 tooling (for example **P17**).

6 Assessment of the modeling solution

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

6.1 Basic modeling constructs

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

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

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

Table 1: Requirements satisfaction for base metamodel

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

Table 2: Requirements satisfaction for Acme software engineering process

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 diagraming TA). The use of slots has been described in the section 4 and is illustrated by bold circles in the Figure 8.

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.

6.3 Number of levels

Due to the fact that our approach is level-agnostic, our solution could have had more or fewer levels depending on the variations of the use case. However, the number of levels is related to the problem, making the approach fixed.

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 each other in a transparent way, using inheritance or instantiation.

6.5 Cross-level constraints

As for cross-level relationships, cross-level constraints do not have a specific nature. They are like any constraint 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 tooling 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. Note that in our approach, the metamodels are built in an ad hoc way together with the models (co-construction), therefore the user also validates on an ad hoc basis.

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 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 apply it 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 ability of our solution: it was decomposed into two steps (Xsure process, then 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 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 tooling 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 tooling 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. Thus, a part of the verification relies on the fact that users add behaviors when they build the metamodels and the models.

7 Related work

A plethora of multi-level modeling approaches and tools with different foundations have appeared in recent years[6] . 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.

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.

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

Rodríguez and Macías 2019 use MultiEcore (Macias Gomez de Villar et al. 2016) to provide a solution to the challenge. MultEcore uses an extension of the two-level cascading technique and potency. They do not need the synthetic typing relations we introduce. However they need to do frequent model transformations to transform instance models into instantiable models. With respect to the proposed solution, we use fewer models as we do not include a generic software engineering process model nor a generic assurance process model.

More similar to us, Jeusfeld 2019 use 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 and 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*.

Similarly, 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). 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 tooling 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.

As a future work we envision to explore a number of alternative solutions: first, we intend to propose another solution based on a more level-agnostic approach such as Clabjects. Instead of two levels defined by Type and Instance concepts, we could have a single concept mixing a type part and an instance part. This approach 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 (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

Atkinson C., Gerbig R. (2016) Flexible deep modeling with melanee. In: Modellierung 2016-Workshopband

Atkinson C., Kühne T. (2001) The essence of multilevel metamodeling. In: International Conference on the Unified Modeling Language. Springer, pp. 19–33

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 Proc. of the Intern. Conf. on Modularity. ACM, 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. ACM, Saint-Malo, France, pp. 24–34 http://doi.acm.org/10.1145/2976767.2976807

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

Lara J. D., Guerra E. (2018) Refactoring multi-level models. In: ACM Transactions on Software Engineering and Methodology (TOSEM) 27(4), pp. 1–56

Macias Gomez de Villar F., Rutle A., Stolz V. (2016) MultEcore: Combining the best of fixed-level and multilevel metamodelling. In: CEUR Workshop Proceedings

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 (Dec. 2016) Semantic Modeling for Information Federation (SIMF). https://www.omg.org/cgi-bin/doc.cgi?ad/2011-12-10. Last Access: May 7th, 2021

OMG (June 2013) OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1. http://www.omg.org/spec/MOF/2.4.1

Openflexo (2019) Openflexo Project. https://www.openflexo.org/. Last Access: May 7th, 2021

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). IEEE, pp. 152–163

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

Steinberg D., Budinsky F., Merks E., Paternostro M. (2008) EMF: eclipse modeling framework. Pearson Education

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