Formalisation et exécutabilité du langage "EditionPattern" au sein de la plate-forme Openflexo

Medhi Alaoui Belghiti

Master Thesis in Computer Science, *June 6, 2013*

Formalisation et exécutabilité du language "EditionPattern" au sein de la plate-forme Openflexo

ALAOUI BELGHITI Mehdi
Master Research of University Rennes1 [1]

SUPERVISOR: Antoine Beugnard

Co-SUPERVISOR: Fabien Dagnat

Department of Computer Science, Telecom Bretagne [2]
PASS(Processes for Adaptive Software System)

[1] http://master.irisa.fr
[2] http://telecom-bretagne.eu

Abstract

Conception of complex systems involves many stakeholders, where each stakeholder models the target system from a particular viewpoint. However, existent modeling paradigms are "stove-piped", which means the information is difficult to share between models produced by these stakeholders, for several reasons. Semantic integration of different viewpoints is the cornerstone of collaborative modeling, because the rational behind modeling is not only representing abstractions of systems, but also generating executable artifacts. Inconsistencies among different views of a system cannot lead to this purpose.

Openflexo is an opensource modeling tool dedicated to collaborative modeling. We propose, in this document, an architecture for promoting semantics sharing between models in Openflexo.

Contents

1 Introduction

In this section, we introduce the notions "Model", "Metamodel", "Domain-Specific Modeling", and "Collaborative Modeling", in order to contextualize our research work, and also, for making the reader familiar with some recurrent concepts, cited many times along this document.

1.1 "Model" notion

A model is an abstraction of a system allowing predictions or inferences to be made [19]. We mean by a system, every interacting or independent elements forming an integrated whole. This opened definition better matches with the wide use of modeling as a transverse activity in many domains (engineering, biology, mathematics, ...). It is evident that a model cannot be a perfect copy of the original system, but the ability to reflect only relevant details of the original is a required quality for models. A model needs to be, also, usable in place of the original system in respect to some purpose.

Abstraction consists of selecting relevant elements of the system under study, needed for a certain purpose, and matching them with some concepts. These concepts are immaterial, so we need a modeling language to represent them with concrete symbols. The Ullmann's triangle 1 explains the relationship between the thing in reality, its abstraction and its symbolic representation.

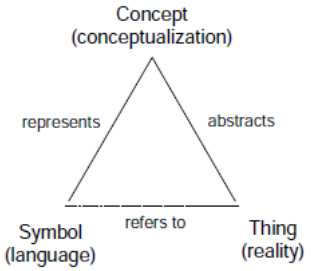


Figure 1: Ulmann's triangle [17]

Much has already been written on features of models [10]. Achieving a consensus about the scope of this notion is urgent, even if "nobody can just define what a model is, and expect that other people will accept this definition; endless discussions have proven that there is no consistent common understanding of models"[20].

1.2 Metamodeling

In its broadest sense, a metamodel is a model of a model. The term "meta" means transcending or above, emphasizing the fact that a metamodel is at higher abstraction than the model itself, which is an abstraction of the system under study. The OMG (Object Management Group) considers that "a model is an instance of a metamodel, meaning that every element of the model is an instance of an element in the metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models"[22]. The figure 2 shows the four layers architecture that synthesize the view of OMG on "metamodel" notion.

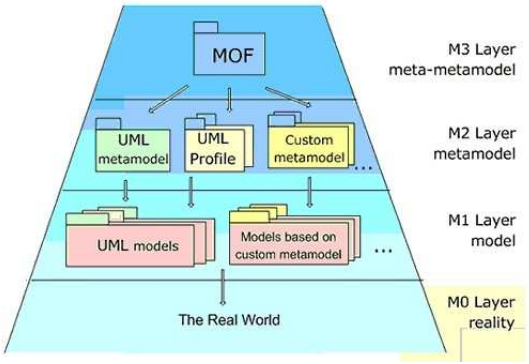


Figure 2: Metamodeling architecture of OMG [22]

The lower level of this hierarchy is M0, contains the executable instances of models elements defined in M1 layer. In M3 level resides the MOF (Meta-Object Facility), which contains some concepts defined in UML. MOF is reflexive since it is used to define itself. All MOF-based metamodels (UML, SysML, CWM, SPEM, etc) are instances of it.

The "metaness" was a subject of [19], where the author think that the vision of OMG is very limited. He proposes two kinds of instantiation that may take place between a model and its metamodel : Ontological and linguistic instantiations. In fact, Ontological instantiation between two elements or models is therefore based on a relationship between them in terms of their meaning. However, a linguistic instantiation defines a syntactic conformance rules between the elements of the two levels. Most of the model elements of the diagram 3 are instances of two types – they are instances of an abstract syntax element (their linguistic type) and a "logical" instance of another model element in the diagram (their ontological type).

Best practices of OMG for dealing with this issue is to use UML profile, which consists of using stereotypes to extend existing UML concepts in order to modify their semantics. However, this solution is partial because the mechanism of type/instance, inherited from a historical attachment of OMG to object-oriented programming, leads to many problems [8]. The most serious problem is "duplication of concepts" in the same level, that is clearly illustrated in figure 4.

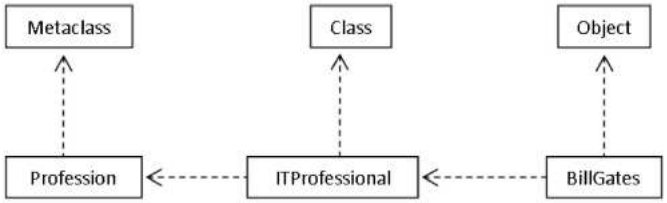


Figure 3: Dual classification [8]

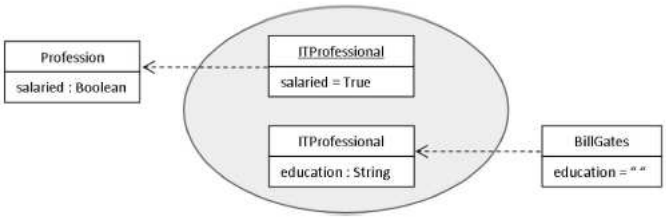


Figure 4: Facets type/instance of one concept [8]

C. Atkinson and T. Kühne [8] propose to abandon the traditional instanciation (type/instance), and promote a multi-level modeling paradigm, called deep metamodeling, based on the "Clabject" : a unification of the "Class" and "Object" facets in the same notation.

1.3 Domain specific Modeling

Domain-Specific Modeling is now a largely adopted approach in the Model Driven Engineering (MDE) community. It enables domain experts to represent themselves their knowledge through a domain-specific notation. Even if UML was primarily designed to be a general-purpose language, it has evolved into UML profile, that enable tailoring model elements using stereotypes to make domain specific customization of the basic UML language. Today, domain-specific modeling is a well-recognized research domain with its own community and conferences such as the OOPSLA Domain-Specific Modeling workshops [5]. As a consequence, we can find several DSML tools in the market, including dependent of UML (such as GMF) and fully independent of UML (such as Microsoft Software Factorie, MetaEdit+, and GME). On the plus side [16], these tools accelerate the requirements engineering process, the users can better understand the models because they are represented in domain-specific terms, the restriction of semantic scope of modeling language (the number of semantics variations to deal with is small compared with general-purpose languages), hence a better support for generating implementations from models.

One of the main problems of domain-specific modeling languages is the "Tower of Babel" [16][8] problem in which the proliferation of different languages reduces the value of models as communication vehicles. If the burden of defining a new DSML becomes considerably lower, we will probably see a large number of DSMLs created and used. The situation will be similar to the one faced with the use of XML-Dialects : Many Dialects will appear and

their interoperability or translation between them will be a major problem.

1.4 Collaborative modeling

Collaborative modeling refers to various tasks relying on various areas of expertise where a number of people actively contribute to the modeling of the same system. The diversity of these tasks leads to a use of different domain-specific modeling tools and languages. As a result, we obtain partial and separately specified models. Each model is a particular viewpoint on the target system, but the sum of these viewpoints does not represent a global abstraction of this system. Managing the consistency of these perspectives remain the cornerstone of collaborative modeling, and a burden for the stakeholders who have to identify contradictions between their models, as it was reported in a field studies [12] [2] of the empiric methods used to manage these issues.

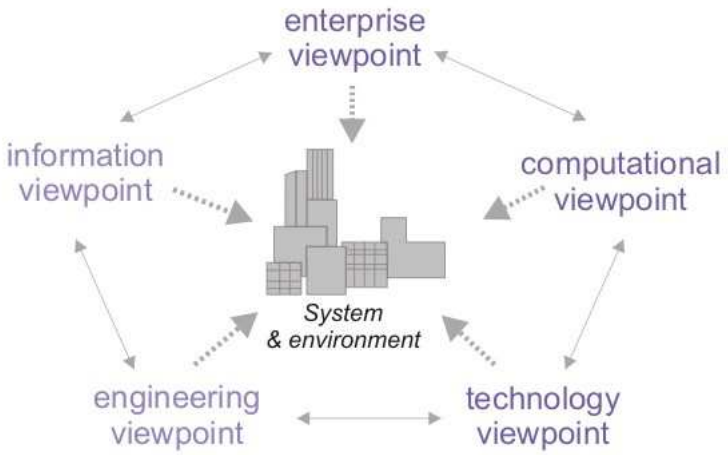


Figure 5: Multi-viewpoint modeling

Architecture description practices, as described in ISO/IEC/IEEE 42010 : 2011[6] standard, utilize multiple views to address several areas of concerns, each one focusing on a specific aspect of the system. Examples of architecture frameworks using multiple views include the Zachman Framework, TOGAF, DoDAF and, RM-ODP. The main concepts of this standard are described in the conceptual model in figure 6 :

- **Architecture description :** Work product used to express fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.
- **Concern :** Interest in a system relevant to one or more of its stakeholders.
- **Stakeholder :** Individual, team, organization, or classes thereof, having an interest in a system.

- **View :** Work product expressing the architecture of a system from the perspective of specific system concerns.
- **Viewpoint :** Work product establishing the conventions for the construction, interpretation and use of architecture views to frame specific system concerns.

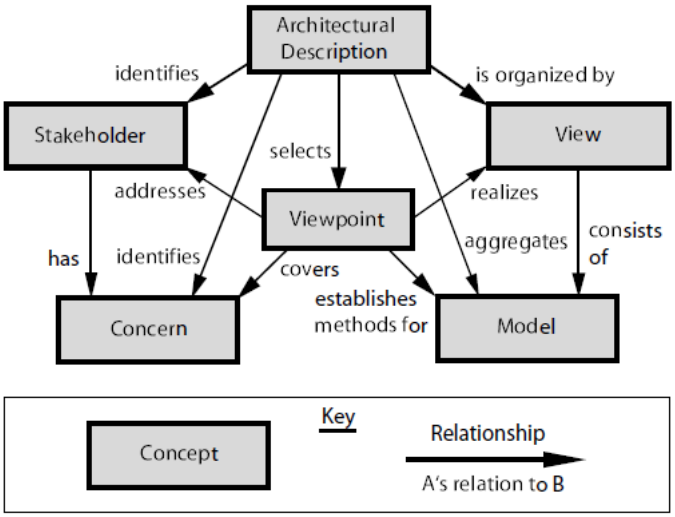


Figure 6: Main concepts of ISO/IEC/IEEE 42010:2011 standard

Other technical challenges in collaborative modeling are "Model integration" and "Version Control"[9]. Model integration (merge) is the procedure through which various models developed by different modeling groups are incorporated in order to obtain a final unified representative view of all the models. A version control system allows developers to maintain multiple copies of their developed models in an organized manner with the ability to incrementally version and number each of the model editions.

This report presents a work on Openflexo, a framework for Collaborative modeling. This tool provides a multi-viewpoint architecture, and enables domain-specific modeling languages tailoring. The purpose of this work is providing an architecture, for Openflexo, that promote semantic integration of models created by different stakeholders in different modeling paradigms. In the next section we analyze Openflexo viewpoint architecture and its main features, then we propose a solution for semantic integration of models and a declarative language for describing elements of this solution.

2 Openflexo : a framework for collaborative modeling

Openflexo is an open-source (GPL3) business architecture platform that supports collaborative modeling, and enables transforming enterprise models into business oriented deliverable. It is evolving from a mature modeling workbench targeted to Business Process Modeling, toward a full-featured infrastructure supporting collaborative multi-paradigm and viewpoint modeling.

Openflexo facilitates the modeling of multiple viewpoints, by formally associating concepts defined in various and heterogeneous metamodels (OWL, EMF, XSD,..) with graphical representation rules and basic actions to manipulate these concepts and their graphical representations in a domain-specific diagram (a view). In addition, Openflexo also provides the user with a set of tools to manipulate these views (graphical editors, textual editors for Domain Specific Language (DSL), tabular editors, ...) but also to use the model for automatically generating artifacts (documents, configuration files, code, ...).

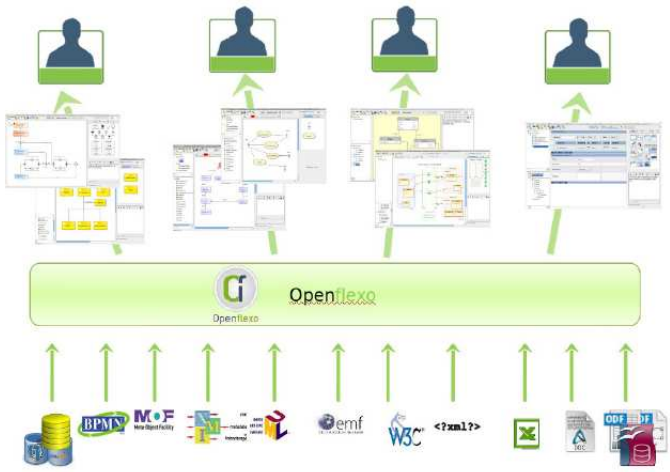


Figure 7: Openflexo : Multi-paradigm and Multi-viewpoint modeling tool

Openflexo is not limited to business usage, it offers a very general approach to complex system modeling through its Viewpoint Architecture. In this document, we focus on this architecture, the use of Openflexo as a Business Process Modeling tool is not addressed.

2.1 Openflexo Viewpoint Architecture

Our modeling framework has been architected (see figure 8) around three sets of non-overlapping concerns: (center) an open and free conceptual modeling space, (left) the designer interactions, (right) projections in the technological or information spaces.

The Information space : Is the collection of databases, files and tools where the information is stored. In this context, any source of information is considered as a model that can be exposed to the Concept Space. This includes the whole set of modeling contents (models and metamodels) and languages that have been defined in many business contexts, using various supporting technologies. Each technology defines a **Technological Space**.

The Concept space : Is the actual core of Openflexo, it provides the user with a flexible environment where he can model easily his abstract concepts. A single unifying abstraction is offered : the EditionPattern (EP). A VirtualModel (VM) provides ways for gathering concepts (EditionPatterns). A VM can be used as a concept itself and as a tool for building new ones. The Conceptual space is populated by EditionPatterns and instantiated using data from the Information Space. The ModelSlot (MS) is an object symbolizing the connection between the different spaces.

The Design space : Is oriented toward the user. Its aim is to support a human-centric approach to information capture and analysis. The modeling workbench can be tailored for each use-case: available features, GUI layouts, look and feel, and modeling paradigms are adapted to the user's needs and habits. From the user's point of view, this level defines Viewpoint models that partition the set of stakeholder's concerns. They provide the interactions, conventions, rules and modeling technologies for constructing, presenting and analyzing custom Views. Viewpoint models can share (partially or completely) the same Concept Space without requiring that each user sticks to the same representation. This ability will ease the reconciliation between stakeholders perspectives.

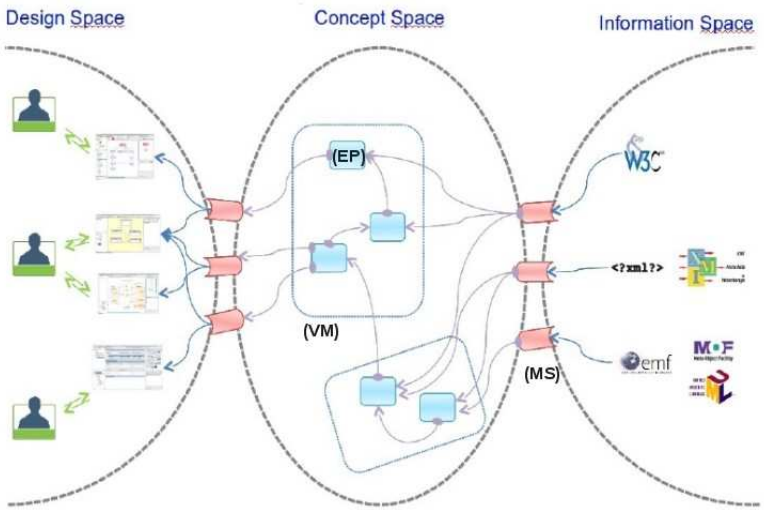


Figure 8: Illustration of the framework's architecture

2.2 The main concepts

The EMF diagram of classes (Appendix A), captures the main concepts of the architecture above and relationships between them. We are going to

give a definition of each concept of this diagram, and then, we are going to analyze, in an example, the whole architecture in respect of the services it renders.

EditionPattern : It refers to the abstraction of a concept, independently from its underlying models and technologies. This abstraction may be represented and encoded differently in many models and technologies. An EditionPattern is not necessarily associated with a particular concept from a model and it is even more often associated with many modeling elements from different models. A PatternRole is an abstraction of the manipulation roles played in the EditionPattern by modelling element potentially in different metamodels.

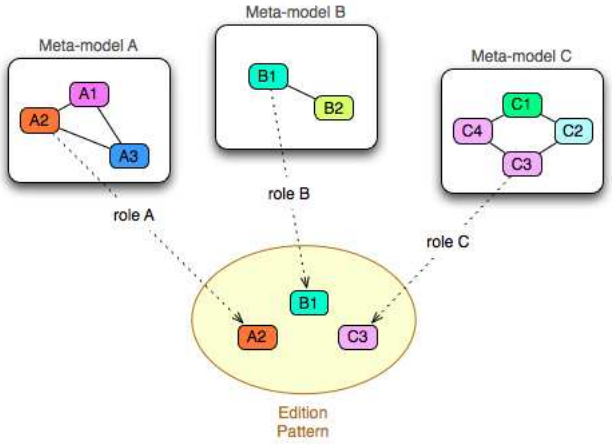


Figure 9: An EditionPattern formed from 3 concepts (A2, B1,C3) defined in 3 different metamodels

EditionSchemes : As said earlier, one of the main objectives of the Openflexo Viewpoint Architecture is to enable the intuitive manipulation of complex model concepts with Openflexo tools (Graphical User Interface (GUI), domain-specific language editors, ...), EditionSchemes are the manipulation primitives defining how data can be created, deleted, accessed (navigation), transformed, etc... Their definition provides something like an API to manipulate a subset of model concepts in the domain-specific context of the EditionPattern.

View/Viewpoint : In the Openflexo Viewpoint Architecture a Viewpoint partitions the set concerns of the stakeholders so that issues related to such concern subsets can be addressed separately. Viewpoints provide the convention, rules and modeling technologies for constructing, presenting and analysing Views. It can address one or several existing models or metamodels. Viewpoints also propose dedicated tools for presenting and manipulating data in the particular context of some stakeholder's concerns. An Openflexo View is the instantiation of a particular Viewpoint with its own Objective relevant to some of the concerns of the Viewpoint.

VirtualModel : Is a modeling component defined in a Viewpoint and instantiated in a View. It is also a container of EditionPatterns and gives a way to instantiate them as EditionPatternInstances, in the VirtualModelInstance. In the class diagram (of Appendix A), we can remark that the class "VirtualModel" is a subclass of "EditionPattern". The objective is to enable the specification of an abstract concept (EditionPattern) as a set of many abstract concepts (EditionPatterns).

ModelSlot : Is used in the Viewpoints to abstract the links to models and metamodels from different technologies. It is a way to hide the complexity of the Technology Adapter used to parse artifacts encoded in different formats.

2.3 Use case

The goal of this use case is to demystify the concepts defined above. Let's take the two metamodels (model 1 and Model 2) of Appendix B. The user can define a Viewpoint, where he can create a Virtual Model as a container of his domain-specific concepts, embodied in EditionPatterns and associated with some elements of these metamodels. Two ModelSlots have to be defined in the user's Viewpoint, to insure the access to these metamodels stored in information spaces, through technological adapters.

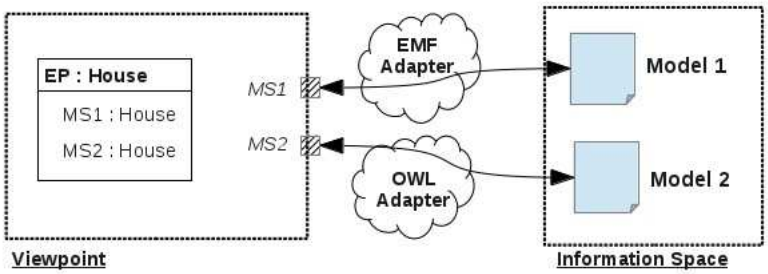


Figure 10: Use case

In the example (figure 10), the user created an EditionPattern that refers to the concept "House". This EditionPattern aggregates two modeling elements, the OWL class "House" from model 1 and the EMF class "House" from model2.

The user can build a friendly tooling above conceptual layer provided by Virtual Model, by associating EditionPattern with some symbolic representations (shapes) and manipulation primitives (EditionSchemes), to construct pallets from which he can drag and drop these elements in Views.

3 A shift to a semantic integration of models

As it was mentioned in section 1.4, the semantic discord between different viewpoints in collaborative modeling tools remain a serious issue. Out of model driven engineering, the semantic integration of information (in general) coming from different sources is still a challenge for several reasons. Semantics is what the data means, and we are not very good at all at understanding how the semantics of data in independent data sources is related. In fact, the stakeholders do not use the same tools and data formats, they do not have the same interpretation of the same data, and the process of coordination between them is complicated. This issue hampers the efficiency of modelers.

The purpose of our work is to propose a new architecture, for our framework Openflexo, that support semantic integration of models stored in the information space.

A basic principle of our approach is that there is no one "right" conceptualization or representation of information. In fact, a concept can be encoded in different formats in the information space, and a data can be interpreted differently in the conceptual space. So we need to be able to define our conceptualizations and representations accurately and then relate them. The figure 11 illustrates the connection between data and concepts.

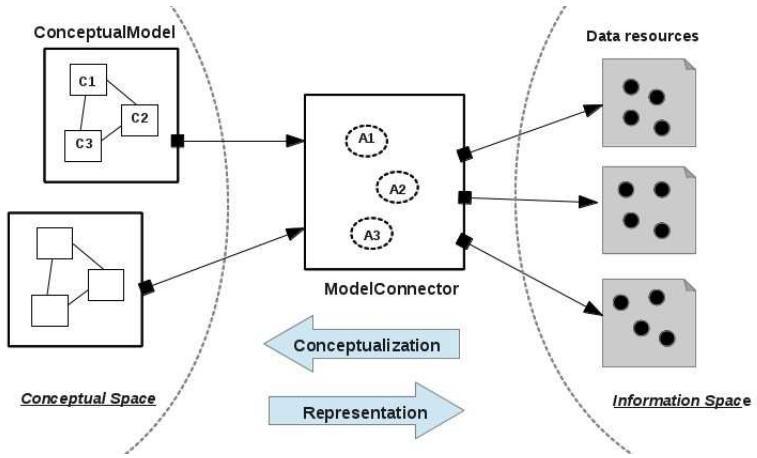


Figure 11: global view on the proposed architecture

The conceptual space is a set of modules called "Conceptual Models", where each module captures the terms and concepts of a subject area. Elements of these conceptual models (called simply "Concepts"), may be encoded differently in different data resources. Semantic relations between concepts define whether they represent the same or related concepts and also help to define the semantics of each concept.

"Model Connector" defines a bridge between the conceptual and information spaces. Elements of these Model Connectors, called "Abstract Elements", define a way to represent a concept in the information space. Each

Abstract Element (dashed circles in figure 11) is associated with a data element. Abstract Element is in full communication with the data element. This communication is enabled through an Event-Action system, where the abstract element detect the changing of the data state, and specify a set of CRUD functions that enable the edition of the data. In brief, we can say that an abstract element is a persistent object related to a potential representation of a concept in a data resource.

The notion of "Concept" is equivalent to "EditionPattern" of Openflexo viewpoint architecture, and the notion "Abstract Element" is slightly equivalent to "PatternRole". We decided to change the used terminology because it was attributed for other purposes, and it does not match anymore with the new problematic that we try to resolve. In this architecture we have held some notions of the previous one, but it is completely different.

It is not our expectation that we can solve this problem 100%, but we aim at proposing the fundamentals of a future solution that can reduce the semantic friction between models. Along this section, we define the architectural elements of our approach. We present also the EBNF (Extended Backus-Naur Form) grammar of a declarative language for the description of these elements.

3.1 Model Connector

Model Connector is a component that specifies an interface in full communication with data elements of the information space, that are relevant to an end user. It gives the possibility to declare a system of expected events related to changing of these data. It is used, also, to specify some CRUD (Create, Read, Update, Delete) functions, called actions, to edit data elements.

The first class citizens of Model Connectors are "Abstract Elements". As its name indicates, an Abstract Element is an abstraction of a data from its underlying technology. Our aim is to align heterogeneous data, coming from different sources, in an intermediate level between the Concepts and Data space. Abstract Elements are run-time objects, utilized to associate a data element to a prospective concept. We will discover the mechanism of this association, called conceptualization, in section 3.2.

A Model Connector needs to be connected with data resources, through Model Slots. We have held the notion of "Model Slot" from the Viewpoint Architecture. In fact, each Model Slot is a symbolic link to a particular data source, and hide the complexity of the technological Adapter that parse the data. The number of Model Slots, declared in the scope of a connector, is equal to the number of data sources form which the user wants to match data with his domain-specific concepts.

In Figure 12, we can recognize the elements of the Model Connector. The hashed square represent a Model Slot that specifies a connection of the Model Connector to an EMF model stored in the Information Space.

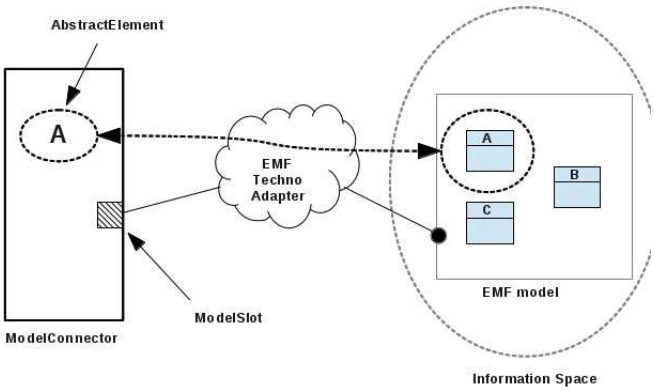


Figure 12: Model Connector elements

This connection is established through an EMF Technology Adapter. The dashed circle is an Abstract Element, named "A" and associated to a Class "A" of the EMF model. The Abstract Element "A" and the Class "A" are fully synchronized, in a way that all events occurring at the class "A" are detected by the Abstract Element.

3.1.1 "ModelConnector" syntax

A Model Connector is declared with the key word "ModelConnector" followed by an identifier. Then between brackets we declare components of the Model Connector (Abstract Elements and Model Slots). The first example, figure 13, shows our first declaration of a Model Connector, named MyConnector, and contains a Model Slot referring to EMF Model, stored in a local folder.

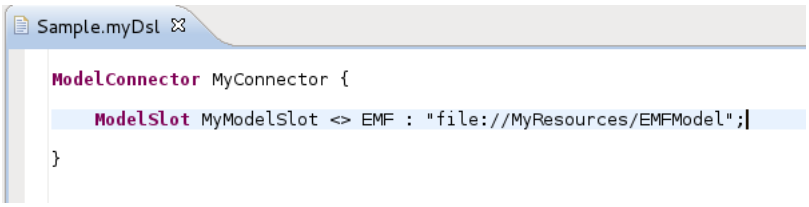


Figure 13: A "Model Connector" declaration (in an editor generated by xtext)

The grammar of a Model Slot declaration looks as follows :

```
ModelSlot:
        'ModelSlot' name=ID '<>' type= TECHNO
        ':' uri=URI ';'
        ;
```

Three information are required to define a Model Slot :

- Its identifier.
- The type of the technology Adapter.
- The URI (Uniform Resource Identifier) of the target data resource.

Until now, there is only three supported technologies : OWL, EMF, XSD. So the type of the technology adapter can take only one these three values :

```
terminal TECHNO:
        "EMF" | "OWL" | "XSD";
```

3.1.2 Abstract Element syntax

A model connector can contain many Abstract Elements, where each such element is characterized by Actions and Events. The syntax of an Abstract Element looks as follows :

```
AbstractElement:
        'AbstractElement' name=ID
        ('<>' slot=[ModelSlot] '::' query=Query)?
        '{'
           elements+=ActEven*
        '}';
```

The basic Abstract Element declaration requires only to define its identifier, in the case where we do not know yet to which data element it should be associated. For example :

```
AbstractElement A {

        }
```

For further precision, we can associate our Abstract Element to a data, by specifying the Model Slot used to access to the resource where the data was stores, and a query to locate this data inside this resource. As it is illustrated in figure 12, we can declare the Abstract Element "A" as follows :

```
AbstractElement A <> MyModelSlot ::
          Select "a:EClass"
          where "a.name='A'" { }
```

We have not already specified the language of querying data resources, but we estimate that the query syntax would be as follows :

```
Query :
    'Select' type=DataType
    ('where' filter=Condition)?
;
```

This simple expression contains two parts : the data type and a condition. The data type represents simply the linguistic classification of an element in a model. It depends strictly of the technology used to store the model. For example in EMF technology we can find the following types (Eclass,

Eattribute, Etype, Ereference, ...). To identify a specific element we specify an additional condition that identify the selected element uniquely. This condition is a constraint on certain properties of the element. The figure 14 shows some properties of an EMF Class.

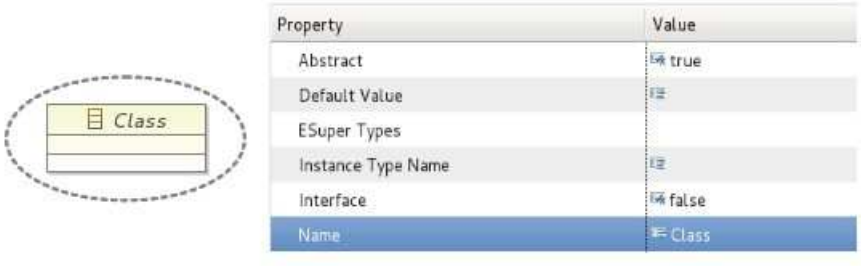


Figure 14: Properties of an EMF class

The possibility to query (EMF, OWL and XSD) models is already feasible in Openflexo, through the software user interfaces (see tutorials [3]). We need to develop a formal language for that purpose.

Once the Abstract Element is declared we should declare a set of Events and Actions that enable interactions between the information space and the conceptual space.

3.1.3 Events

The information space is an active environment, because its content can change, due to the use of data resources by external tools. An event can be defined as "a significant change" in a data element. For example, a user can change the name of the class "A", of the EMF model figure 12, to "B". Another user, who has defined a conceptual model which depends on this class, may treat this changing as an event because he considers that the name of a class is semantically significant. From a formal perspective, what is created, deleted and updated in the information space triggers a message called Event.

Our event system is illustrated in figure 15. This system is based on three entities :

- **A control**, which is the event source in the information space.
- **Interfaces**, which are Abstract Elements, describe the events that have to be communicated.
- **The clients**, which are federated concepts in the concepts space. They receive the events from the interfaces and they react according to specified rules.

[3] `www.openflexo.org`

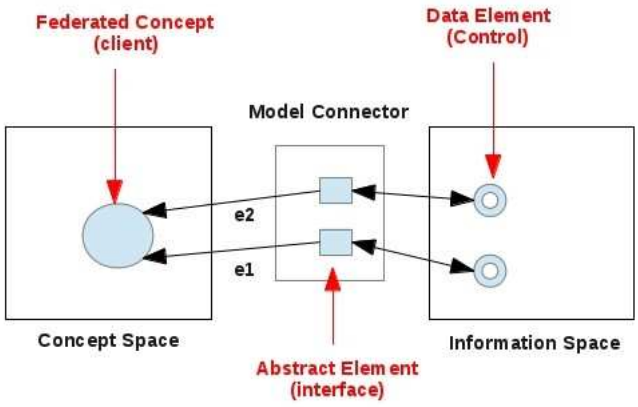


Figure 15: Event system

We are going to see the reactive aspect of our event system in section 3.2, i.e. the behavior of federated concepts after receiving events. In this part we focus on the specification of Events that have to be detected by the interfaces.

There are some basic events that we consider to be important in the context of this architecture. We list these events :

- isCreated : immediately after a data element is created.
- isDeleted : immediately after a data element is deleted.
- isUpdated : immediately after a data element is updated.

```
terminal BASICEVENT:
        "isDeleted" | "isCreated" | "isUpdated"
;
```

We can associate a basic event with a logical condition, to obtain a logical event. For example, a user can be interested to the event resulted from an update of the class "A" and the fact that it is not changed to an abstract class after this update. This event can be specified as follows :

```
event e = (isUpdated & "A.Abstract = false");
```

The grammar of logical events is specified below. Every basic event, by default, a logical event.

```
LogicalEvent :
        "(" + BASICEVENT + '&' + Condition")"
        |  BASICEVENT
;
```

Logical events can be combined to create composite events using logic operators. For example, a user can be interested to the event resulted from an update of the class “A”, or its suppress. We can formulate this event with the expression :

```
event e = [isUpdated | isDeleted];
```

The grammar of composite events looks as follows :

```
CompositeEvent :
  "[" + LogicalEvent +
  (LOGICSYMBOLE + CompositeEvent)*"]"
  | LogicalEvent
;
terminal LOGICSYMBOLE:
        "&" | "|" | "^"
;
```

3.1.4 Actions

Model Connector is also an interface that include functions such as Read, Create, Update, and Delete. These functions are called Actions. Each action is used to edit a property of a data element. You can see in the figure 14 an example of an EMF class properties.

“Read” operations include all operations that return the value of a property. For example, we can define an action to read the name of the EMF class A of the example figure 12. The keyword “this” refers to the current Abstract Element.

```
Action getName() = Read "this.name";
```

More than that, we can specify actions to read properties of elements that compose a data. For example, we suppose that the class “A” has an attribute called “alpha”. We can specify an action for reading this attribute as follows :

```
Action getAttributeType() = Read "a.EType"
       Select "a:this.EAttribute"
       Where "a.name='alpha'" ;
```

The expression that follows “Select” is a description of the element, that we want to know its type. The language used in this expression is still dependent on the technology of the data resource (“Etype” and “Eattribute” are keywords in EMF technology). As we have said earlier, it is interesting to look for a way to free our language from technological aspects.

We can imagine a scenario where a user define a concept in the conceptual space, but there is no data in information space that match with this concept. He could think, for example, to create a class “Z” in an EMF model

to represent this concept. “Create” actions are used to create new elements in the information space. The user can define this action as follows :

```
Action newClass() = Create "a:Eclass"
                    with "a.name = 'Z'";
```

3.2 Conceptual Model

A “Conceptual Model” captures the concepts of a subject area – it is a model of a domain or business. These concepts are used in more than one model of the Information Space. Some conceptual models can use or depend on each others but there is no expectation that any module represent the “universal truth” about anything. In brief, a conceptual model is a certain interpretation of a set of data by one or a group of stakeholders.

A conceptual model is a set of entities, called “Concepts”, each Concept aggregates one or many Abstract Elements. The mapping between concepts and data is carried out at Model Connector level. An abstract element represent a certain way to store or to “persist” a concept for a particular use. The meaning of a Concept is limited to the scope of one Conceptual Model.

Each concept, in the scope of one Conceptual Model, is identifiable with its name. There is no support of polysemy in our architecture : every entity of a conceptual model means one and only one thing. However, it is possible to have equivalent concepts with different names. Except this condition, our conceptual space is an “open world” : anyone can say anything about anything, in any conceptual model.

A Conceptual Model is not a fixed entity, but it is expected to evolve over time. In fact each entity of a Conceptual Model is synchronized with a data element of the information space. These data, which could be created for external user’s purposes, can be subject of modification to keep synchronization with multiple problem domain. We mean by “external user”, a user who does not take part in the creation of Conceptual Model and use a data resource for his own purposes.

To deal with this changing of data resources we propose a reactive behavior, called “Dwhen”, and used to exploit the Actions and Event declared in one Model Connector to impose some constraints that should be verified between data elements.

In this section, we analyze the syntactic elements of Conceptual Models and their role in the semantic integration of models architecture.

3.2.1 “Conceptual Model” Syntax

The grammar of Conceptual Model declaration is the following :

```
ConceptualModel:
        'ConceptualModel' name=ID '{'
        elements+=ConceptualModelComponent*
        '}';

ConceptualModelComponent:
        Concept | Use
;
```

A Conceptual Model is identifiable with a name, and contains :

- "use" expressions : utilized to declare the Model Connectors used in this Conceptual Model as bridges to information space.
- "concept" : an object that embodies a certain idea.

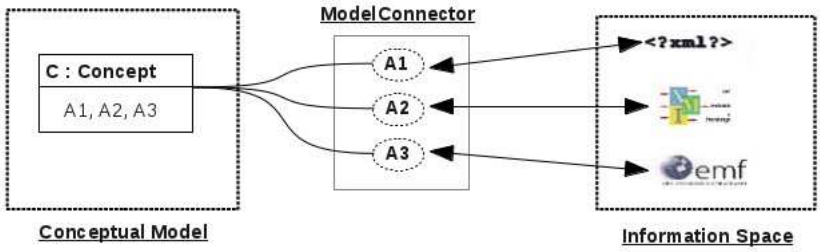


Figure 16: A concept aggregates three Abstract Elements

For example, we can start by declaring a conceptual model, named MyConceptualModel, which use the model connector, MyConnector, that we have created in the previous subsection :

```
ConceptualModel MyConceptualModel {
        use MyConnector;
}
```

The syntax used to declare a concept is the following :

```
Concept :
        'Concept' name=ID '{'
            elements+=ConceptElement*
        '}';

ConceptElement :
    Conceptualize | Dwhen |
    SubclassOf | SynonymOf | Association
    InstanceOf
    ;
```

A concept has an identifier and contains elements among which we have specified until now :

- "Conceptualize" expressions
- "Dwhen" expressions
- Semantic links : "SubclassOf", "SynonymOf", "Association"
- Ontological instantiation : "InstanceOf"

3.2.2 "Conceptualize" syntax

We have chosen the keyword "conceptulize" to declare which abstract elements are associated with one concept. "To conceptualize" means [4] to form a concept or concepts of, and especially to interpret in a conceptual way, and this is exactly what we do. We form a concept by interpreting a set of abstractions of independent data (Abstract Elements), specified in a Model Connector. The grammar of "Conceptualize" expressions is the following :

```
Conceptualize :
        'conceptualize' variable=STRING
           '=' concept+=[OneConceptualize]    ';'
;

OneConceptualize :
        concept+= [ModelConnector] + "."
              + (concept+=[AbstractElement])
;
```

After the keyword 'conceptualize' the user should affect to a variable a reference to an Abstract Element. We can associate the Abstract Element "A" that we have declared in the previous subsection with a new concept, named "C", as follows :

```
ConceptualModel MyConceptualModel {
        use ModelConnector;
        Concept C {
                conceptualize a = ModelConnector.A;
        }
}
```

3.2.3 "DWhen" syntax

In a concept, users can define expressions that enforce constraints between the data stored in information space designated as reactive. The term "reactive" means that any modification performed on these data may trigger one or multiple Actions in order to maintain a certain rule.

We call these expressions "Dwhen", a contraction of "Do When". This keyword synthesizes the reactive function that we want to specify.

[4] http://www.thefreedictionary.com/conceptualize

The specified actions after "Do" are automatically executed whenever the Event after the "When" is detected. This event is specified in one of the abstract Elements, associated to the concept in which this expression figures. We are going to see some examples of the use of such these expressions in the use case (part 3-3).

Reactive programming is interesting for models federation, but it suppose that the user know well which actions he should execute to solve a constraint. It is not always evident to declare "Dwhen" expressions because we can fall in contradictions. The following example illustrates a possible contradiction :

```
(1) Do : A.delete()  when : e();
(2) Do : A.update() when : e() ;
```

The first DWhen statement leads to a suppress of the data A, just after the event e(), meanwhile, the data A should be updated, according to the second statement. The situation is conflicting because we do not know which action is going to precede. The user have to avoid such these pitfalls by reasoning on priorities and dependencies of DWhen expressions to insure consistency.

A system of exceptions is also necessary. We can imagine a scenario where the event e() (in the second Dwhen statement) occurs but the data which is associated with the Abstract Element A does not exist. In this case, the Action "update" cannot be executed, so this error should be reported to the user.

3.2.4 "SubclassOf"

In a Conceptual Model, we need to establish hierarchical relationships between concepts. A concept "A" is a subclass of a concept "B" if the concept "B" is more general and include the meaning of "A". For example the concept "Human" is a subclass of "Mammal". This classification is expressed in our language by the mean of the keyword "SubclassOf":

```
Concept Human {
                SubclassOf Mammal;
        }
```

In our approach, we authorize multiple subsomption. In fact, a concept can be a subclass of more than one concept. For example,in figure 17, the concept "Professor" is a subclass of "Person" and "Employee".

```
Concept Professor {
                SubclassOf Person, Employee;
        }
```

The grammar of "SubclassOf" expressions :

```
SubclassOf:
        'SubclassOf' concept+=[Concept]
```

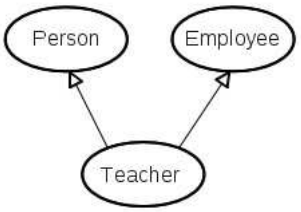


Figure 17: Example of subsomption

```
                    (',' + concept+=[Concept])*  ';'
;
```

3.2.5 "SynonymOf"

Two concepts in a Conceptual Model can be synonyms, which means, they have the same meaning. For example, the creator of a Conceptual Model, can create these there concepts "Vehicle", "Voiture" and "Carriage", and consider them as synonyms :

```
Concept Vehicle {
                SynonymOf Voiture, Carriage;
        }
```

3.2.6 Association

In a Conceptual Model, we need to know how concepts are semantically related. An association is a semantic link between two concepts. Each concept can contain many associations that have names, and point on another concepts. In this document, we talk only about binary associations. For example, the concept "Teacher" is associated with the concepts "School" and "Course". We can declare this association as follows :

```
Concept Teacher {
            Association "work in" : School;
            Association "teaches" : Course;

        }
```

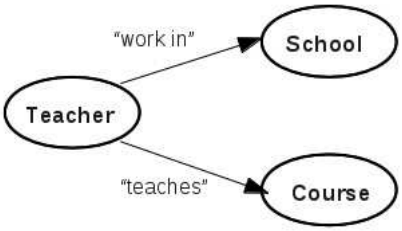


Figure 18: Example of association

The grammar of "Association" statements looks as follows :

```
Association:
    'Association' name=STRING ':' concept=[Concept] ';'
;
```

3.2.7 instantiation

One of the most basic concepts of modeling is "types". The concept of types only makes sense in the presence of "instances" of those types. The concept of "type" is best described as a "predicate"; something true about the instance. I could have a type named "blue square" and as long as I can test that something is a square and that it is blue, I can determine if something is a blue square.

The concept class in object programming does not test instances, it creates them and through the language design that guarantees that something created as a square stays a square. So an OO class is a factory for instances where the predicate will always be true – the predicate is that the instance is created by the class. Ontology languages (like OWL) allow you to assert that an instance is of some type, it will also infer a type if it is required by the logic.

As we have said earlier (in subsection 1.2), there are two kinds of instantiation : Ontological and linguistic. The first one confirms that a type is something true about an instance in term of its meaning. In the conceptual model, we focus on describing this relation between concepts by the mean of the keyword "InstanceOf". A user can assert that a concept is an instance of many other concepts.

The Conceptual Model should support multiple instantiation, i.e., the ability of a concept to be conform to more than one type. For example, I am a student and a trainee, so the concept of "Me" is an instance of the concept "Student" and "Trainee".

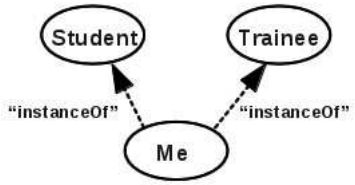


Figure 19: Example of multi-instantiation

```
Concept Me {
                InstanceOf Student, Trainee
        }
```

In our approach we try to break rules of "Strict Metamodeling", adopted by OMG in the four layers metamodeling architecture. "Strict Metamodeling" asserts that every element of a level Mn have exactly one type in the level Mn+1, and the only relationship that can cross two successive levels is instantiation. As we have said earlier (in subsection 1.2), this rule hampers dramatically the potential of metamodeling.

We have already broken this rule by our support to multiple instantiation which is supported by ontological modeling languages such as OWL. The solution of "deep metamodeling" and "powertype" were proposed among others. We need to make a state of art of Multilevel modeling approaches to uncover a real solution more compatible with our situation.

3.3 Use case

In this section we present an application of our approach to the use case of Appendix B, where we have three models (model1, model2, model3). We suppose that these models are stored in the information space in different formats (EMF, OWL and Excel). We suppose also that we have tools to access to these different formats, i.e. technology adapters.

These three models contain some common concepts. We have to demonstrate how we could represent dependencies between these concepts in a Conceptual Model.

3.3.1 Model Connector

First we have to create a Model Connector as a bridge between the information and conceptual spaces. This Model Connector should contain three Model Slots representing a symbolic access to models. We can declare this Model Connector with the name "MC" as follows :

```
ModelConnector MC {
    ModelSlot Model1 <> EMF : "URI://Model1";
    ModelSlot Model2 <> OWL : "URI://Model2";
    ModelSlot Model3 <> Excel : "URI://Model3";
}
```

In these three data resources we have the concept "City" encoded in different ways. This concept is represented by classes in Model1 and Model2, named "City". In Model3 the user considers that the column B represents the concept city. In our Model Connector, we should relate these representations with Abstract Elements :

```
ModelConnector MC {
  AbstractElement EMFCity <> Model1 ::
          Select "a:EClass"
          Where "a.name = City"  {}

  AbstractElement OWLCity <> Model2 ::
         Select "a:OWLClass"
```

```
            Where "a.name = City" {}

    AbstractElement ExcelCity <> Model3 ::
            Select "a:Column"
            Where "a.name = B" {}
}
```

3.3.2 Conceptual Model

A user can create some concepts, related with the Abstract Elements declared in the Model Connector, in a Conceptual Model. We can create our Conceptual Model that use the ModelConnector (described above) as follows :

```
ConceptualModel {
      use MC;
}
```

In this Conceptual Model, we create the first concept "City" that federate the three representations of this concept in the information space, which are described in Abstract Element :

```
ConceptualModel {
      use MC;
      Concept City {
            conceptualize a = MC.EMFCity;
            conceptualize b= MC.OWLCity;
            conceptualize c= MC.ExcelCity;
      }
}
```

3.3.3 Reactive Behavior

We suppose that the user wants to relate the name of the class city in Model1 with the name of the class city in Model2. If the name of the first one change, the name of the second should take the same value. The user should declare some actions and events in Abstract Elements related with this data. First we declare an event in the Abstract Element "EMFCity" to detect the update of the class "city" in Model1, and an action "getName" to read the new name :

```
ModelConnector MC {
    AbstractElement EMFCity <> Model1 ::
            Select "a:EClass"
            Where "a.name = City"  {
                  Event e = isUpdated;
                  Action getName() = Read "this.name"
            }
}
```

We should declare also an action “setName” in the Abstract Element “OWLCity” to update the name of the class “City” in Model2 :

```
ModelConnector MC {
    AbstractElement OWLCity <> Model2 ::
            Select "a:OWLClass"
            Where "a.name = City" {
                Action setName (newName) =
                        Update "this.name" <- newName
            }
}
```

In the Conceptual Model we declare the reactive behavior that relate between these actions and event :

```
ConceptualModel {
      use MC;
      Concept City {
            conceptualize a = MC.EMFCity;
            conceptualize b= MC.OWLCity;
            conceptualize c= MC.ExcelCity;
            Do b.setName(a.getName()) When a.e
      }
}
```

3.3.4 Semantic links

The user can declare many Abstract Elements associated with elements of the three models. In the Conceptual Model, the user can create some Concepts related to these Abstract Elements. In Appendix B we have created some Abstract Elements and Concepts. We have also related these concepts semantically using the language specified in this document. The figure 20 illustrates the relationships between concepts as it is specified in the Conceptual Model (see Appendix B).

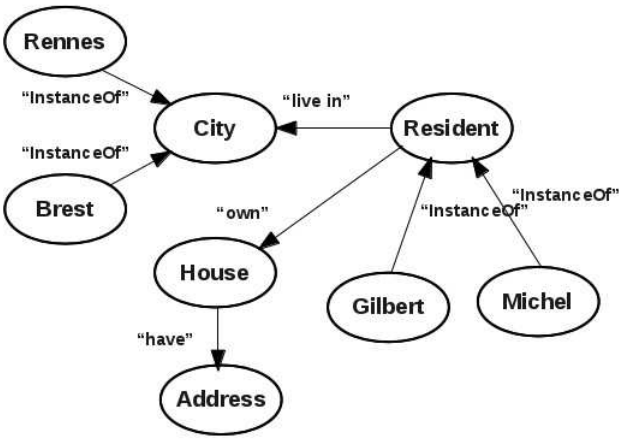


Figure 20: Graphical representation of the conceptual model of use case

4 Related works and conclusion

Our Work is based on SIMF (Semantic Information for Modeling Federation) request for proposal [23]. OMG asks for submissions for a standard that addresses the federation of information across different representations, levels of abstraction, communities, organizations, viewpoints, and authorities. The theory of SIMF is that we can better relate the semantics and format of data in logical information models, as it is illustrated in the architecture of figure 21. SIMF relies on the existent standards to conceive at least one graphical and one textual notation for expressing the elements of this architecture. We have adapted some notions in SIMF theory to our context, that is Openflexo framework, aiming at proposing an executable environment and a textual language for information federation in collaborative modeling.

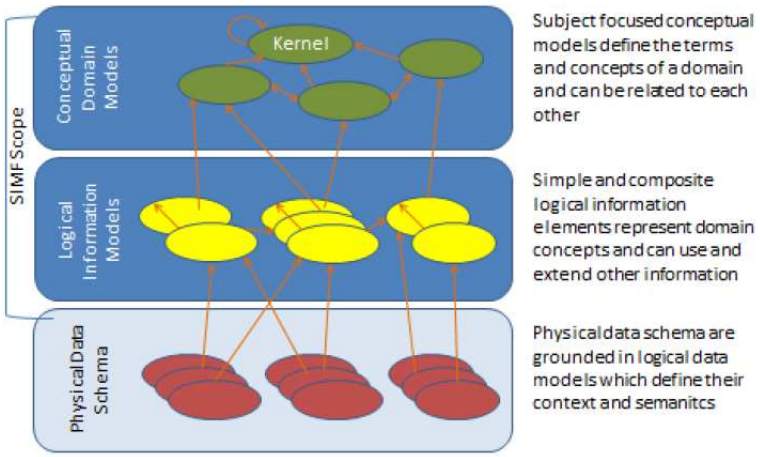


Figure 21: Properties of an EMF class

We are interested in architectural practices for designing open distributed systems define several distinct viewpoints. In the literature we can find a plethora of frameworks such as the "4+1" view model [1], OpenViews [11] or Dijkman's framework [14]. Most of these frameworks do not consider correspondences between viewpoints, or assume they are trivially based on name equality between correspondent elements. We are interested also in some approaches that deal with the problem of inconsistency among viewpoints (see examples [15] , [4], [3]).

Querying and transforming models are primordial activities in our approach. We are interested in languages that are specially developed for this purpose such as QVT [21], ATL [7] and other languages used for specific technologies like XSLT for XML based models and SPARQL for ontologies. Comparisons [18] [24] between these languages are also relevant to our work.

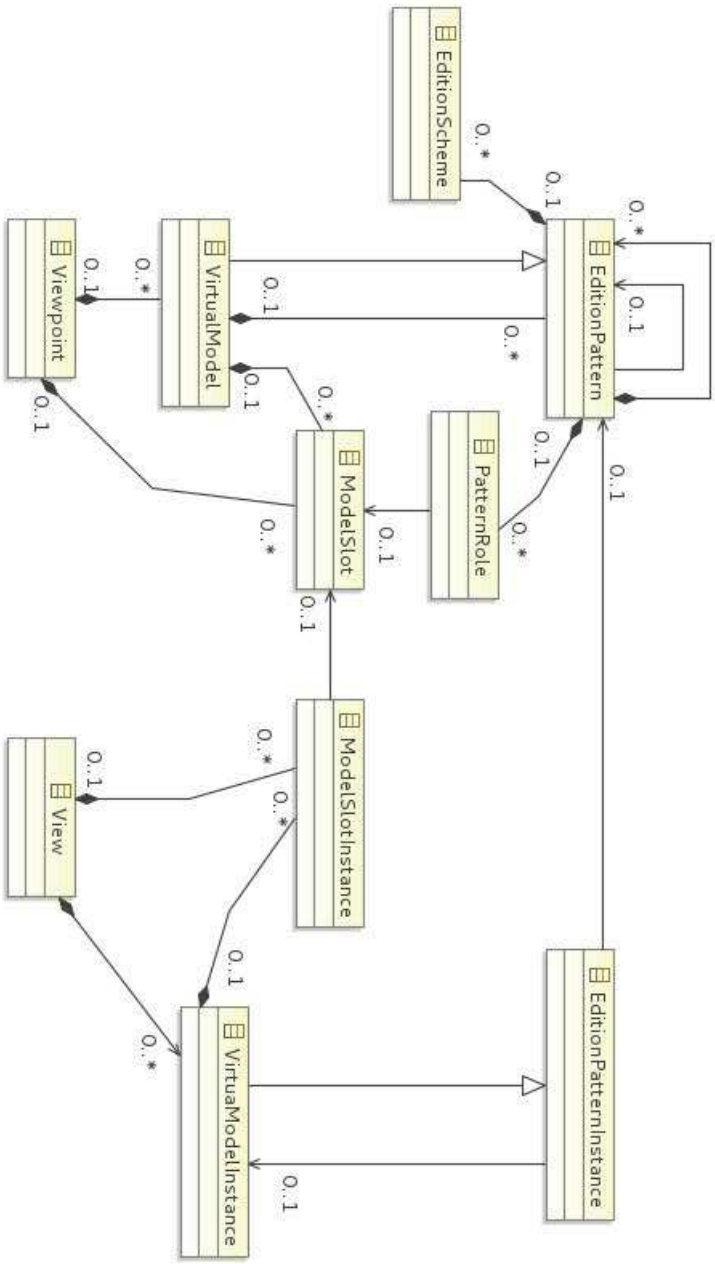
Maintaining constraints between models elements lead us to dig into many programming paradigms. We have proposed, in the first place, reactive

programming [13], but we need a deep state of art regarding the theory of reactive languages.

This report described our work on the problematic of semantic integration of models. We have presented the elements of the architecture and the grammar of a textual language to describe these elements. However, this proposition is not a mature solution, we have to work on a deep state of art of the different topics described above.

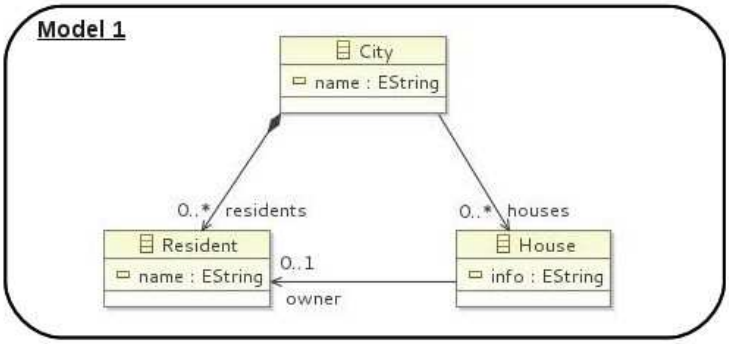
Appendices

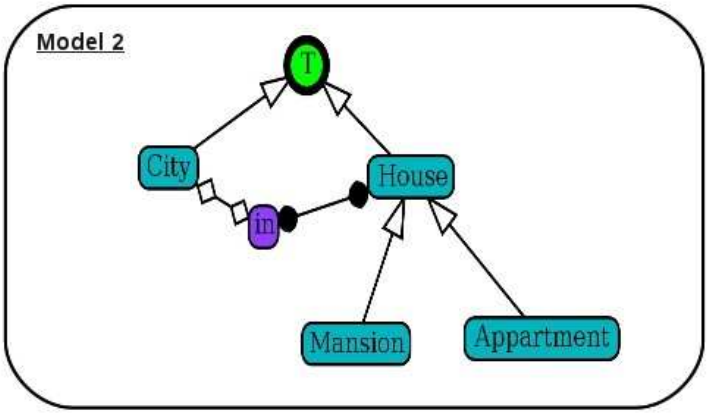
A The conceptual model of Openflexo Viewpoint Architecture

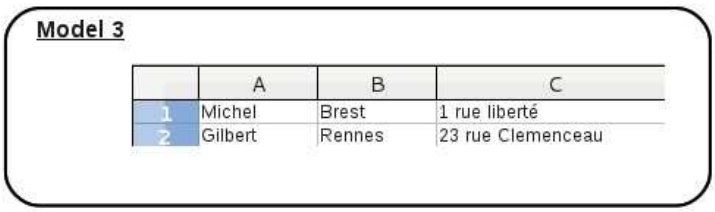


B Use case

B.1 three Models







B.2 Model Connector

```
ModelConnector MC {
    AbstractElement EMFCity <> Model1 ::
            Select "a:EClass"
            Where "a.name = City" {}

    AbstractElement OWLCity <> Model2 ::
            Select "a:OWLClass"
            Where "a.name = City" {}

    AbstractElement ExcelCity <> Model3 ::
            Select "a:Column"
            Where "a.name = B" {}
```

```
    AbstractElement Brest <> Model3 ::
            Select "a:Cell"
            Where "a.(Column, Row) = (B,1)" {}

    AbstractElement Rennes <> Model3 ::
            Select "a:Cell"
            Where "a.(Column, Row) = (B,2)" {}

    AbstractElement EMFResident <> Model1 ::
             Select "a:EClass"
             Where "a.name = Resident"  {}

    AbstractElement ExcelResident <> Model3 ::
            Select "a:Column"
            Where "a.name = A" {}

    AbstractElement Michel <> Model3 ::
            Select "a:Cell"
            Where "a.(Column, Row) = (A,1)" {}

    AbstractElement Gilbert <> Model3 ::
            Select "a:Cell"
            Where "a.(Column, Row) = (A,2)" {}

    AbstractElement EMFHouse <> Model1 ::
             Select "a:EClass"
             Where "a.name = House"  {}

    AbstractElement OWLHouse <> Model2 ::
            Select "a:OWLClass"
            Where "a.name = House" {}

    AbstractElement EMFAddress <> Model1 ::
             Select "a:EAttribute"
             Where "a.EClass = House"
             And "a.name = info"  {}

    AbstractElement ExcelAddress <> Model3 ::
            Select "a:Column"
            Where "a.name = B" {}
}
```

B.3 Conceptual Model

```
ConceptualModel  MyConceptualModel{
      use MC;
      Concept City {
            conceptualize a = MC.EMFCity;
```

```
        conceptualize b= MC.OWLCity;
        conceptualize c= MC.ExcelCity;
        Do b.setName(a.getName()) When a.e
    }

    Concept Resident {
         conceptualize a= MC.EMFResident;
         conceptualize b= MC.ExcelResident;
         Association "own" : House
         Association "live in" : City
}

    Concept House {
        conceptualize a = MC.EMFHouse;
        conceptualize b= MC.OWLHouse;
        Association "have" : Address
    }

    Concept Address {
        conceptualize a = MC.EMFAddress;
        conceptualize c= MC.ExcelAddress;
    }

    Concept Michel{
        conceptualize b= MC.Michel;
        InstanceOf Resident
    }

    Concept Gilbert{
        conceptualize b= MC.Gilbert;
        InstanceOf Resident
    }

    Concept Brest{
        conceptualize b= MC.Brest;
        InstanceOf City;
    }

    Concept Rennes{
        conceptualize b= MC.Rennes;
        InstanceOf City;
    }
```

References

[1] Architecture Blueprints - the "4+1" View Model of Software Architecture, 1995.

[2] An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering. IEEE Computer Society Press, 1999.

[3] Fixing Inconsistencies in UML Design Models, ICSE '07, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Realizing Correspondences in Multi-viewpoint Specifications, 2009.

[5] Proc. 12th OOPSLA Workshop Domain-Specific Modeling, University of Alabama, United States, August 2012.

[6] ISO/IEC/IEEE 42010:2011. Systems and software engineering and architecture description. 2011. `http://cabibbo.dia.uniroma3.it/asw/altrui/iso-iec-ieee-42010-2011.pdf`, last visite 19-05-2013.

[7] Freddy Allilaire and Tarik Idrissi. Adt: Eclipse development tools for atl. In Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2), pages 171–178. Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK, 2004.

[8] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. IEEE Transactions on Software Engineering, 35(6):742–755, 2009.

[9] Ebrahim Bagheri and Ali A. Ghorbani. An exploratory classification of applications in the realm of collaborative modeling and design. Inf. Syst. E-Business Management, 8(3):257–286, 2010.

[10] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. Upgrade, 5(2):21–24, April 2004.

[11] Eerke A. Boiten, Howard Bowman, John Derrick, Peter F. Linington, and Maarten Steen. Viewpoint consistency in odp. Computer Networks, 34(3):503–537, 2000.

[12] Moises Branco, Yingfei Xiong, Krzysztof Czarnecki, Jochen M. Küster, and Hagen Voelzer. An empirical study on consistency management of business and it process models. 2012.

[13] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. SIGPLAN Not., 46(10):407–426, October 2011.

[14] Remco M. Dijkman, Dick A. C. Quartel, and Marten J. Van Sinderen. Consistency in multi-viewpoint architectural design. 2006.

[15] Steve Easterbrook and Bashar Nuseibeh. Using viewpoints for inconsistency management. SOFTWARE ENGINEERING, 11:31–43, 1996.

[16] Robert B. France and Bernhard Rumpe. Domain specific modeling. Software and System Modeling, 4(1):1–3, 2005.

[17] Giancarlo Guizzardi. On ontology, ontologies, conceptualizations, modeling languages, and (meta)models. Proceedings of the 2007 conference on Databases and Information Systems IV: Selected Papers from the Seventh International Baltic Conference DB&IS'2006, pages 18–39, 2007.

[18] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track, 2006.

[19] Thomas Kühne. Matters of (meta-)modeling. Software and System Modeling, 5(4):369–385, 2006.

[20] Jochen Ludewig. Models in software engineering. Software and System Modeling, 2(1):5–14, 2003.

[21] OMG. MOF QVT Final Adopted Specification, 2005. OMG doc. ptc/05-11-01.

[22] OMG. Omg unified modeling language (uml) infrastructure, version 2.4.1. 2011. `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PS`, last visite 19-05-2013.

[23] OMG. Semantic information modeling for federation (simf) rfp. may 2012. `http://www.omg.org/cgi-bin/doc?ad/11-12-10`, dernière visite 09-01-2012.

[24] Marcel van Amstel, Steven Bosems, Ivan Kurtev, and Luís Ferreira Pires. Performance in model transformations: Experiments with atl and qvt. In ICMT, pages 198–212, 2011.