# Handling domain knowledge in system design models. An ontology based approach.

Kahina Hacid


# THÈSE

**En vue de l'obtention du**

# DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

**Délivré par :**

Institut National Polytechnique de Toulouse (INP Toulouse)

**Discipline ou spécialité :**

Sureté de Logiciel et Calcul à Haute Performance

---

## Présentée et soutenue par :

Mme KAHINA HACID

le mardi 6 mars 2018

## Titre :

Handling domain knowledge in system design models. An ontology based approach.

---

**Ecole doctorale :**

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

**Unité de recherche :**

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

**Directeur(s) de Thèse :**

M. YAMINE AIT AMEUR

**Rapporteurs :**

M. ANTOINE BEUGNARD, TELECOM BRETAGNE CAMPUS DE BREST

Mme REGINE LALEAU, UNIVERSITE PARIS 12

**Membre(s) du jury :**

M. DOMINIQUE MERY, UNIVERSITÉ LORRAINE, Président

M. LAURENT ZIMMER, DASSAULT AVIATION, Membre

M. YAMINE AIT AMEUR, INP TOULOUSE, Membre

# Abstract

Complex systems models are designed in heterogeneous domains and this heterogeneity is rarely considered explicitly when describing and validating processes. Moreover, these systems usually involve several domain experts and several design models corresponding to different analyses (views) of the same system. However, no explicit information regarding the characteristics neither of the domain nor of the performed system analyses is given.

In our thesis, we propose a general framework for complex systems models strengthening and multi-view analysis. We offer first, means for the formalization of domain knowledge using ontologies and second, the capability to strengthen design models by making explicit references to the domain knowledge formalized in these ontology. Finally, this framework provides resources for making explicit the features of an analysis by formalizing them within models qualified as "points of view".

We have set up two deployments of our approach: a Model Driven Engineering (MDE) based deployment and a formal methods one based on proof and refinement.

The general framework has been applied to several case studies issued from engineering domain. Prototypes corresponding to the deployment of our approach in the MDE and Formal methods settings have been developed, and deployed. Experiments with MDE based techniques have been conducted on the particular engineering domain of avionic systems.

The work presented in this thesis has been developed as part of the AME Corac-Panda project and ANR-IMPEX project.

# Acknowledgments

I would like to address my thanks to all the people who have helped me and have contributed to the completion of this thesis.

First, I would like to thank my PhD director Yamine Aït-Ameur for offering me a PhD position on such an interesting research issues. His guidance helped me in the time of research and writing of this thesis. I would also like to thank him for leading and connecting me with ANR-IMPEX and AME-CORAC PANDA research projects that dealt with issues of high scientific interest which undoubtedly enriched my work and my research.

I would like to sincerely thank Regine Laleau and Antoine Beugnard who have reviewed this thesis. Thank you for having carefully examined my work and provided valuable feedback and constructive suggestions. I also thank them as well as Dominique Méry and Laurent Zimmer for accepting to be part of my defense committee.

I would like to thank the colleagues with whom I was pleased to daily evolve and work with. To the PhD students colleagues I would like to say that work would just not have been same without all the good times we spent together. I thank Guillaume in particular for his guidance during my first months as a PhD student. I sincerely thank Soukayna, Mathieu, Sarah and Alexandra for their kindness, support and help. I am very grateful to you four.

Many thanks to Sylvie Armengaud and Annabelle Sansus. Thanks to their efficient and scrupulous work, I have been able to enjoy my work at ENSEEIHT without worrying about scholar administrative problems.

I want to express my deep gratitude to my family for their support and the constant encouragement I have received from them over the years. Especially my parents who always have been extremely supportive of me and who have made countless sacrifices all along my life to help me get to this point. Thank you all for your faith in me. I undoubtedly could not have done this without you and words cannot express how grateful I am to you.

Last but not least, I would like to thank Adel for his unfailing support. He has been on my side all along my PhD. I deeply value his implication and his belief in me. I can not thank you enough for your constant encouragement, your patience and all the precious advice that helped me getting through so many difficult and stressful times.

# Contents

# List of Figures

# Part I

# Context

# Introduction

As part of the system engineering and complex system design, the models designed by engineers are placed at the center of the development process of the understudied system. Engineers use them to describe, reason, analyze, simulate, animate, validate, verify, etc. systems operating in different environments, domains and contexts. In addition, these models correspond to partial *views* of the studied system (e.g. functional, real-time, energy, mechanics, reliability, architecture, etc.). This engineering development process leads to the production of several heterogeneous models addressing the same system. These models are designed to handle specific analyses. They are qualified as *"design models"*.

In this context, the most important heterogeneity factors, in addition to the one of the modeling languages, are those related to information, knowledge and assumptions of the underlying studied domain (environment and context of implementation and execution of the designed systems) that are implicitly considered by this engineering development process. Indeed, domain knowledge information is usually not explicitly handled and therefore not explicitly included in the models associated to the system under design that may be a critical system.

Moreover, modeling languages are not equipped with operators (or mechanisms) that may support domain knowledge modeling. In fact, although these models are developed in accordance with standards and good practices, an important amount of knowledge, required for model interpretation and validation, remain implicit. As a consequence, a system may be considered as correct with respect to the initial requirements but, it can miss some of its relevant properties if the information related to its application domain are not handled in the design model. A simple illustrative example would be considering a system which allows the execution of an addition of two integer variables $X$ and $Y$ (i.e $X+Y$), which can be invalidated if information from its application domain states that $X$ is measured in *meters* and $Y$ in *miles* is made explicit.

Finally, in most of the developments, there is no explicit information regarding the characteristics of the performed system analyses. In fact, the system developer usually uses only part of the system model for a specific activity (an analysis to be performed on the model) and thus, some concepts of the design model may be not useful for a given analysis. For example, real-time analysis does not require all the functional concepts of the analyzed system. There is no explicit definition of the concepts of a given model required by a given model analysis. Moreover, when performing model analysis, the system developer does not explicitly describe the analysis that he/she used. In order to take design decisions, another system developer needs the whole information and hypotheses related to the realized system model analyses. Indeed, the result of an analysis may interfere with the input and the results of another analysis. Thus information regarding the used method, tool, properties for an analysis may be needed to best evaluate its corresponding output result. The system developer needs to know for example: what are the performed model analyses (tool, method, input, output, etc.)? What are the hypotheses made by the other analyses? And what are the parts of the system that have been analyzed?

# Problem statement

Based on our observation outlined previously, our general research problem issued from engineering design processes can be formulated as:

***How to make explicit domain knowledge in design and multi-view analysis of system design models?***

# Research goals

Taking into account the previous discussion, the objective of our thesis research work is to propose a sound and operationalized approach for design models strengthening and multi-view analysis. Decomposing this overall research objective, we formulate three research goals addressed in our thesis work.

1. Provide a formal framework supporting the explicit modeling of domain knowledge. [***RG1***]

2. Establish an explicit link or reference between domain knowledge models and the design models and thus, express the properties entailed by the domain knowledge reference on the enriched design models. [***RG2***]

3. Make explicit the information regarding the characteristics of system analyses in a multi-view analysis context and report multi-view analyses results in the whole system development. [***RG3***]

# Complex systems design methods

Design models are expressed in different heterogeneous modeling languages (textual, graphical, semi-formal, formal, etc.) and are analyzed using different verification and validation methods (some methods being more formal than others). MDE is a promising approach for the design of complex systems, it allows the modular representation of the system and thus the separation of concerns.

In the following we recall some notions and concepts related to modeling, modeling languages in the context of both MDE and formal methods that we have made extensive use of in our thesis work.

## Model Driven Engineering

MDE brought several significant improvements to the development cycle of complex systems allowing system developers to focus on more abstract levels than classical programming level. MDE is a form of generative engineering [87] in which all or part of an information system is generated from models. A system can be described by several models corresponding to several views or abstraction levels. These models are often described using either graphical or textual notations supported by semi-formal modeling languages. These languages support the description and representation of both structural, descriptive and behavioral aspects of a system. The capability to define constraints that limit the interpretation domain of models is offered using constraints definition languages. In this context, UML[77], the MOF[75] and OCL[78] play the role of standard, they are widely and commonly used by the MDE community.

Moreover, MDE handles models at different development stages of a system development life-cycle. MDE offers several techniques to automate different development steps. Indeed, model operationalization for code generation, documentation, testing, validation, verification, implementation and model analysis are available. These techniques use the capability to transform source models either to other target models in order to get benefits from the available analysis techniques offered by the target modeling technique or to source code in a given programming language. Transformations are defined by means of transformation rules describing the correspondences between the entities in the source model and those of the target model. This transformation

process is automated as much as possible by means of processing programs, which are in most cases developed in general purpose languages (e.g Java) or in dedicated transformation modeling languages (e.g ATL[1], Kermeta[2], QVT [76]).

## State-based formal methods

[94] defines formal methods as *" mathematically-based techniques for describing system properties. They provide frameworks within which people can specify, develop and verify systems in a systematic rather than ad hoc manner"*. In [18] formal methods *"provide a notation for the formal specification of a system whereby the desired properties are described in a way that can be reasoned about, either formally in mathematics or informally but rigorously"*.

State-based formal methods is a category of formal methods where a system is specified in terms of *states* and *transitions* between *states*. *States* are explicitly described using a set of variables and their possible range of values and *transitions* are defined by events formalizing the operations on the system *states*. Pre and post-conditions [52] [31] are usually associated to these events. State-based formal methods can be associated to a refinement mechanism to support an incremental description of a system leading from an abstract level to a more concrete one.

State-based formal methods are usually used to guarantee safety, security reachability, etc. properties within critical systems. Many examples can be found in the literature, we can cite Z[86], VDM[62], TLA[66], B[5], EVENT-B[6]. These methods are associated to tools supports like PROMELA/Spin[53] and Rodin[58], TINA[11], ProB[67] offering formal modeling, verification, animation and model-checking capabilities.

In this thesis, we choose to use the Event-B state-based formal method to model and prove the correctness of our systems. The Event-B method is considered as an evolution of the B method. It is based on set theory and first order logic and its details are given is the next section.

## Event-B formal method

The Event-B method [6] is a stepwise formal correct-by-construction development method. It is based on the refinement of an initial model, a machine, by gradually adding design decisions. A set of proof obligations (PO), based on the weakest

[1]ATLAS Transformation Language: http://www.eclipse.org/atl/
[2]Kermeta: http://www.kermeta.org/

| **CONTEXT** | **MACHINE** |
|---|---|
| *ctxt_id_2* | *machine_id_2* |
| **EXTENDS** | **REFINES** |
| *ctxt_id_1* | *machine_id_1* |
| **SETS** | **SEES** |
| $s$ | *ctxt_id_2* |
| **CONSTANTS** | **VARIABLES** |
| $c$ | $v$ |
| **AXIOMS** | **INVARIANTS** |
| $A(s,c)$ | $I(s,c,v)$ |
| **THEOREMS** | **THEOREMS** |
| $T_c(s,c)$ | $T_m(s,c,v)$ |
| **END** | **VARIANT** |
| | $V(s,c,v)$ |
| | **EVENTS** |
| | **Event** $evt =$ |
| | **any** $x$ |
| | **where** $G(s,c,v,x)$ |
| | **then** |
| | $v :\mid BA(s,c,v,x,v')$ |
| | **end** |
| | **END** |

(a) Contexts and machines.

| Theorems | $A(s,c) \Rightarrow T_c(s,c)$<br>$A(s,c) \wedge I(s,c,v) \Rightarrow T_m(s,c,v)$ |
|---|---|
| Invariant preservation | $A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x)$<br>$\wedge BA(s,c,v,x,v') \Rightarrow I(s,c,v')$ |
| Event feasibility | $A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x)$<br>$\Rightarrow \exists v'.BA(s,c,v,x,v')$ |
| Variant progress | $A(s,c) \wedge I(s,c,v)$<br>$\wedge G(s,c,v,x) \wedge BA(s,c,v,x,v')$<br>$\Rightarrow V(s,c,v') < V(s,c,v)$ |

(b) Generated proof obligations for a model.

Figure 1: Basic definitions: contexts, machines and proof obligations.

precondition calculus[31], is associated to each machine. Development correctness is guaranteed by proving these PO.

An Event-B model [6] (see figure 1(a)) is defined by a *MACHINE*. It encodes a state transitions system which consists of: the variables declared in the *VARIABLES* clause to represent the state; and the events declared in the *EVENTS* clause to represent the transitions (defined by a Before-After predicate BA) from one state to another.

The model holds also *INVARIANTS* and *THEOREMS* to represent relevant properties of the defined model. Then a decreasing *VARIANT* may introduce convergence properties when needed. An Event-B machine is related, through the *SEES* clause to a *CONTEXT* which contains the relevant sets, constants axioms, and theorems needed for defining an Event-B model. The refinement capability [7], introduced by the *REFINES* clause, decomposes a model (thus a transition system) into another transition system with more design decisions while moving from an abstract level to a less abstract one. New variables and new events may be introduced at the refinement level. In a refinement, the invariants shall link the variables of the refined machine with the ones of the refining machine. A gluing invariant is introduced for this purpose. It preserves the proved properties and supports the definition of new ones.
Once an Event-B machine is defined, a set of proof obligations is generated. They are submitted to the embedded prover in the RODIN [6] platform. Here the prime

notation is used to denote the value of a variable after an event is triggered. More details on proof obligations and on the Event-B method can be found in [6].

# Contributions

Our research method is inspired by practice and the modularity decomposition principle has been followed.

1. To address ***RG1***, we have described domain knowledge as theories. Domain ontologies are used for this purpose. Indeed, we believe that ontologies are good candidates for describing and making explicit a domain knowledge[10].

2. A formal explicit link between design models and domain ontologies has been established to address ***RG2***. In fact, annotation of model resources by references to ontologies makes it possible to handle domain knowledge in design models.

3. ***RG3*** is reached through the definition of explicit models that describes the whole features of an analysis, and uses the contribution associated to ***RG2*** to link these analyses to design models.

# Publications

This section gives an overview of the published papers related on thesis work.

- **Paper A.** [44] *Strengthening MDE and formal design models by references to domain ontologies. A model annotation based approach.* **Kahina HACID**, Yamine Ait-Ameur. *7th International Symposium On Leveraging Applications of formal methods, verification and validation*, Corfu, Greece, **Springer**, October 2016.

- **Paper B.** [41] *Handling Domain Knowledge in Formal Design Models: An Ontology Based Approach.* **Kahina HACID**, Yamine Ait-Ameur. *7th International Symposium On Leveraging Applications of formal methods, verification and validation*, Corfu, Greece, **Springer**, October 2016.

- **Paper C.** [45] *Annotation of engineering models by references to domain ontologies.* **Kahina HACID**, Yamine Ait-Ameur. *6th International Conference on Model and Data Engineering* , Almeria, Spain, **Springer**, September 2016.

- **Paper D.** [43] *Handling Domain Knowledge in Design and Analysis of Design Models.* **Kahina HACID**, Yamine Ait-Ameur. *7th International Symposium On Leveraging Applications of formal methods, verification and validation*, Corfu, Greece, **EASST**, *journal paper.*

## Associated projects

Our thesis work has been conducted in the context of two research projects : The AME-CORAC project and ANR-IMPEX project.

- **AME-CORAC project.** [3] *Avionic Modular Elements - COnseil pour la Recherche Aéronautique Civile.* addresses design and validation of an innovative architecture for embedded avionic systems.
- **ANR-IMPEX project.** [4] *Agence Nationale de la Recherche - IMplicitExplicit.* is about integration of implicit and explicit semantics of knowledge domains in a proof based environment. It is based on the use of state-based formal methods and ontologies.

[3] http://aerorecherchecorac.com/
[4] http://www.agence-nationale-recherche.fr/Projet-ANR-13-INSE-0001

# Chapter 1

# Ontologies and domain knowledge

Ontologies have drawn a lot of efforts and interest within the computer-science community. First, a lot of efforts have been devoted to the study of ontologies and their applications in the area of semantic web and information retrieval. Several approaches for describing, designing and formalizing ontologies for these application domains have been proposed by many authors.

In this chapter, we give an overview of domain ontologies , their characteristics and their different domains of use.

## 1.1 Domain ontologies

Gruber defines an ontology as *an explicit specification of a conceptualization* [37]. We consider a domain ontology as a *formal and consensual dictionary of categories and properties of entities of a domain and the relationships that hold among them* [61]. By entity we mean being, i.e, any concept that can be said to be in the domain. The term dictionary emphasizes that any entity of the ontology and any kind of domain relationship described in the domain ontology may be referenced directly, for any purpose and from any context, independently of other entities or relationships, by a symbol. This identification symbol may be either a language independent identifier, or a language-specific set of words. But, whatever this symbol is, and unlike in a linguistic dictionary, this symbol denotes directly a domain entity or relationship, the *description* of which is formally *stated* providing for (automatic) reasoning and consistency checking.

Ontologies have drawn a lot of efforts and interest within the computer-science community. First a lot of efforts have been devoted to the study of ontologies and their applications in the area of semantic web and information retrieval. Several approaches for describing, designing and formalizing ontologies for these application domains

have been proposed by many authors. Models [24, 19, 79, 51, 84, 82], browsers like Protege[1] [65, 1] or PlibEditor[2], reasoners [14, 39, 40, 74], annotators [29, 46], XML-based translators [20, 91] have been developed to engineer such ontologies and establish formal links with the studied domain objects like texts, images, videos, signals, ... Most of these approaches paid a lot of attention to the use of ontologies to interpret these objects and/or provide classifications of these interpreted objects.

### 1.1.1 Some fundamental characteristics

A domain ontology is an explicit conceptualization of domain entities and relationships. As mentioned in [54] [10], ontology definitions will fulfill four fundamental criteria [60].

1. **Formality.** An ontology is a conceptualization with an underlying formal semantics and equipped with reasoning capabilities. It is expressed within a modeling language equipped with a satisfaction relationship ($\models_O$) offering interpretation capabilities (e.g. checking that an instance belongs to the model defined by the ontology) and an entailment relationship ($\vdash_O$) to handle proofs (e.g. proving that a statement can be derived from axioms and theorems defined by the ontology). As a consequence, checking properties expressed over the ontology-defined concepts and individuals becomes possible thanks to the associated theory, either by automatic or semi-automatic reasoning techniques.

2. **Consensuality.** Agreement on the conceptualization defined by an ontology will be reached for a large community of users. This community is not restricted to users or to developers of a specific application: it gathers all the potential users and developers of other applications related to the conceptualized domain. Consequently, an ontology will be shared by several applications and design models. For example, ISO13584-compliant (PLIB) [82, 84] product ontologies are defined according to a formal standardization process. They are published as ISO and/or IEC international standards. This criterion excludes conceptual models defined for a specific application.

3. **Capability to be referenced.** As stated in the previous definition, each concept defined in an ontology is associated to an identifier or *URI* (Uniform Resource Identifier) provided to allow applications to refer this concept from any

[1] http://protege.stanford.edu/
[2] http://www.plib.ensma.fr/

environment. Moreover, this concept can be referenced whatever the ontology model is set up to describe this concept.

4. **Canonical and non canonical.** One other important characteristic is related to the design process. In the case of the engineering domain, ontologies are built from canonical (primitive) concepts, then non canonical (derived) concepts are defined from canonical ones by composition of derivation operators (restriction, union, intersection, algebraic operators, etc.) available in the ontology modeling language. Note that terms are associated to each concept.

### 1.1.2 An example of ontology

Figure 1.1 depicts an extract of the well-known *Pizza* domain ontology example which was developed by the University of Manchester for educational purposes. It formalizes all the domain information related to the *Pizzas* and their topping. *Thing* is the root ontology concept from which every other pizzas concepts (and any other ontology) are subsumed using a *is_a* relationship (inheritance or subsumption relationship). This kind of relationship is used to construct a hierarchy and organize the domain concepts.

The domain knowledge can be split to two abstract concepts: *Pizzas* and *Topping* respectively representing all the existing pizzas and all their possible topping. Domain properties and relationships are also formalized. *hasTopping* property is defined (as a Restriction relationship) for example on the *Margherita* pizza to describe that this kind of pizza is only composed of *some Tomato* and *some Mozzarella* topping.

Domain instances are attached to the formalized pizza classes. For example, *Margarita_domino's* and *Margharita_Hut* are defined as instances of *Margharita*. In the same way, instances of other type of pizzas can be formalized.

Ontologies are descriptive models of a given domain. Thus, they can always evolve. For the *Pizza* ontology example, new kind of pizzas, topping and instances (if new pizza stores open, etc) can be added and described in this ontology.

## 1.2 Ontology modeling languages

Several ontology modeling languages were developed during last ten years, as for example, Ontolingua [33] for artificial intelligence applications, DAML+OIL [24], RDFS [19] and OWL [79, 51] for Web applications, and PLIB [84, 82] for engineering

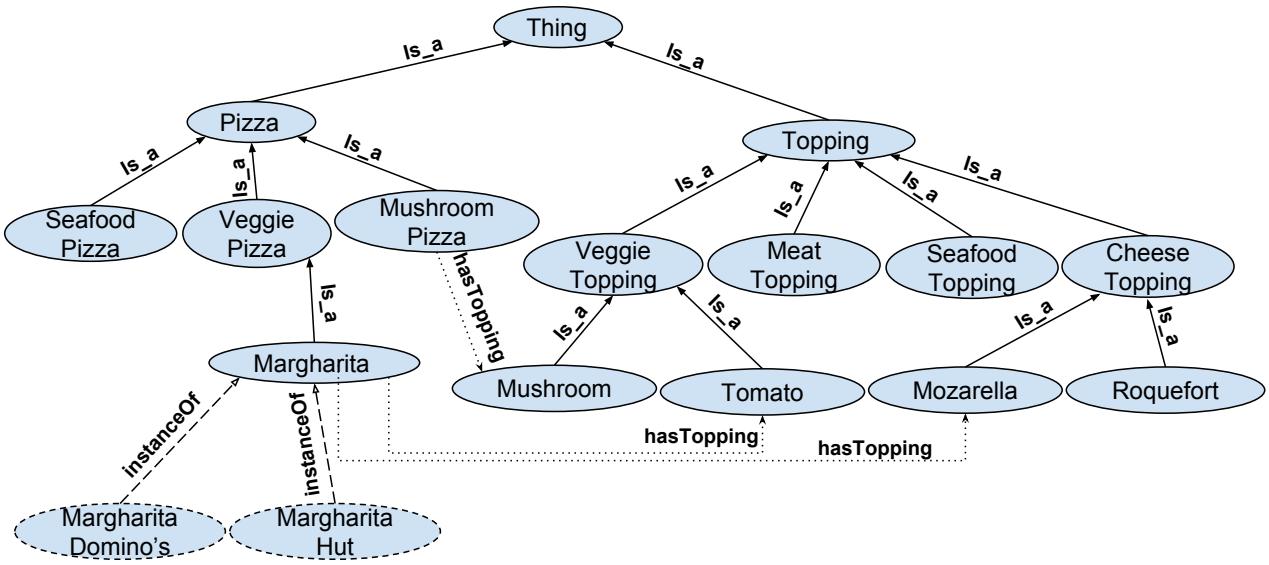



Figure 1.1: The *Pizza* ontology

applications. [73] have developed the OntoEventB plug-in to automatically support the translation of ontologies models, described using ontology description languages such as OWL, PLIB or RDFS, into Event-B Contexts. The shallow modeling approach has been chosen to describe translated ontologies in Event-B.

### 1.2.1 Main ontology modeling languages characteristics

In [32] [54], authors have identified some relevant characteristics associated to ontology modeling languages. These characteristics have been extended with new ones in order to handle the system engineering aspects.

- **Words and concepts.** Ontology modeling languages offer the capability to describe words and concepts. Various relationships are offered by these languages: between words, between concepts and between words and concepts. The presence of such relationships leads to two ontologies design processes [60]: from words to concepts (e.g. semantic web) or from concepts to words (e.g. engineering catalogues).

- **Strong typing.** Ontology modeling languages are equipped with a type system characterizing classes, properties, relationships and domain values. According to the modeling language, this type system may be more or less a strong type system. For example, the PLIB ontology modeling language uses a strong typing system (e.g. unit types are built-in types) while description logics do not require

strong typing. Types are useful for indexation, and thus for the definition of persistent frameworks like semantic databases [21, 28, 80, 49, 81, 89] to store both ontologies and their instances.

- **Algebraic operators** may be associated to the types available in the ontology language. Operators on classes like union, intersection, etc. are available in most of the ontology modeling languages. For example, operators on reals, are available in the PLIB ontology model. They make it possible to express property derivation (e.g. diameter equals two times a radius). These defined algebraic operators define abstract data types and give complete definition of the data types discussed above.

- **Constraints description** constructs are offered by the ontology modeling language to define constraints on classes, properties or on whole ontology. In the engineering domain, the more the constraint description language is rich, the more the expressed concepts of the ontologies are precisely described. Checking these constraints depends on the used constraint solving techniques associated to the ontology modeling language.

- **CWA v.s. OWA.** Closed world assumption (CWA) implies that a complete knowledge is known and, if a fact is not a consequence of the ontology model, then its negation is, while in open world assumption (OWA) this reasoning is no longer available. In general, CWA is used in system engineering, while semantic web considers OWA.

- **Context modeling.** In the engineering domain, the context in which a property is defined is important [83]. At the ontology level, the domain of a context dependent property is not only its class, but it is also a context description (usually a class). For example, the definition of the lifetime (property) of a tyre (class) depends on the average temperature of use (context of use). Note that PLIB offers built-in constructs for such properties.

- **Inheritance and instantiation.** Classes may be linked by single or multiple inheritance relationships. Inheritance helps to factorize objects with the same properties, it also contributes to the definition of the subsumption relationship. Instances of a class represent the individuals, and an individual may belong to a single class (mono-instantiation) or to several classes (multi-instantiation). Ontology modeling languages offer different forms of inheritance and instantiation.

For example OWL supports multiple inheritance and multi-instantiation while PLIB supports single inheritance and mono-instantiation.

- **Reasoning.** In ontology engineering, reasoning essentially concerns subsumption (e.g. to link ontologies classes in case of integration), class membership checking (e.g. for migration of instances from one ontology class to another one) and classification (e.g. to build new class hierarchies according to some criteria). Other logical aspects of reasoning concern the specific properties of the underlying logic like symmetry, reflexivity, equivalence etc are useful for knowledge inference. Different reasoning techniques and tools have been defined [14, 39, 40, 74]. Like for model checking, these approaches may face the problems of memory saturation or space exploration. Other reasoning techniques more commonly used by formal methods have also been set up to handle proof of properties in ontologies. These approaches, which proved scalable, use theorem provers like COQ [12, 27] or Event-B [8] to infer ontologies properties.

- **Exchange formats**. All the ontology modeling languages offer exchange formats based on the XML language. When expressed in this exchange format, classes and their instances can be interpreted in different contexts of use.

## 1.3 Ontologies in engineering

Our work does not address semantic web applications. Our thesis is involved in the engineering area. Thus, we focus on domain ontologies where the whole knowledge on the domain is described in the provided ontology. Due to the system engineering targeted application domain, we use ontologies conforming to the PLIB ontology model [9, 38, 85, 56]. This ontology model advocates the use of strong typing with a rich type system (similar to the one of a programming language specific types like units), property derivation with algebraic operators corresponding to the defined types, first order logic and set theory as a constraint language, CWA and context dependent properties.

Like in usual engineering practices and unlike OWL, additional models may be added to a technical object description. Indeed, a set of different functional models, each one representing a particular view or discipline-specific representation (e.g., safety, real time, energy consumption, geometry procurement, simulation, etc.) can be associated to a given technical object described within the PLIB ontology model.

Finally, a number of domain ontologies based on this model already exist. Examples are the ISO 13584 and ISO 15926 (e.g. mechanical fasteners, measure instruments, cutting tools) and IEC 61360 (e.g. electronic components, process instruments) series of ontologies developed within international standardization organizations (e.g. ISO, IEC) or national ones (e.g. JEMIMA[3] CNIS[4]) that cover progressively all the technical domain.

## 1.4 Ontologies v.s. design models

Ontologies and design models define a conceptualization of a part of the world through the definition of models within modeling languages. From this point of view, an ontology seems similar to a design model [50]. Below we compare design models with respect to the criteria we have identified for ontologies.

Design models respect the *formal* criterion. Indeed, a design model is based on a rigorously formalized logical theory and reasoning is provided by view mechanisms. However, a design model is application requirement driven: it *prescribes* and *imposes* which information will be represented in a particular application (logical model). Indeed, two different information systems have always at least slightly different application requirements, and design models are different from systems to systems. Thus, design models do not fulfill the *consensual* criterion. Moreover, an identifier of a design model defining a concept is a name that can only be referenced unambiguously inside the context of an information system based on this particular design model. Thus, design models also do not fulfill the *capability to be referenced* criterion. Efforts have to be devoted for explicit references to model entities, hence annotations mechanisms are defined. They are discussed in the next section.

## 1.5 Ontologies and annotations

One of the main usage of ontologies is annotation. Let us consider a set of entities available in a given corpus. These entities may be words or sentences in a document, images or videos, entities of a design model, etc. By annotation, we denote the link that may exist between an ontology concept (class, instance, property, etc.) and an entity of the considered corpus. The annotation process consists in defining and running a set of rules leading to the production of annotations. This process may be

[3]Japan Electric Measuring Instruments Manufacturers Association,
[4]Chinese Institute for Standardization

completely automated, semi-automatic with user validation or completely interactive. Automatic annotation proved powerful in the area of the semantic web and natural language processing where the entities of the corpus are words appearing in texts. In [15, 25, 30, 47], the authors use ontologies for raw data annotation in an informal context. Web pages and documents are annotated with semantic information formalized within linguistic ontologies. Once annotations are achieved, formal reasoning is performed. Several tools (or annotators) have been developed for various ontologies and different natural languages [16, 26, 29, 46, 48]. Other approaches to annotate images and multi-media documents have also been developed [23].

In the area of system design, the objective of model annotation is to increase model interoperability. Consensual domain ontologies are shared by different system models corresponding to different engineering views. Annotations allow the designer to link different entities of different system models to ontology concepts. Reasoning at the ontology level makes it possible to check some domain properties. Model annotations have been produced in a semi-formal context using interactive and/or semi-automatic annotation. [13] propose an automated technique for integrating heterogeneous data sources called "ontology-based database". This approach assumes the existence of a shared ontology and guarantees the autonomy of each source by extending the shared ontology to define its local ontology. In [17, 93, 69, 70, 95] annotations are made in an interoperable context and aim to improve the reading, common understanding and re-usability of the models and thus enabling unambiguous exchange of models. In [68], a reasoning phase is performed based on the output of the annotation phase. The reasoning rules produce inference results : (1) Suggestion of semantic annotation, (2) Detection of inconsistencies between semantic annotations and (3) Conflict identification between annotated objects. These approaches addressing interoperability issues focused on improving the common understanding of models. They do not deal with the formal correctness of models with respect to domain properties and constraints.

In the context of formal methods, approaches for semantic enrichment of design models related to an application domain using formal annotations are defined. Annotations are directly set up inside the models. Examples of such approaches are the classical pre and post-conditions of Hoare pre-conditions [52] or program annotation tools like Why3[34]. In [64], the authors introduce real-world types to document the programs with relevant characteristics and constraints of the real-world entities. Real-world types are connected to entities of the programs (variables, functions, etc). The reasoning and checking of the correctness of programs in regards to real-world

types becomes possible by type checking. These approaches seem close to ours, but, to the best of our knowledge, they do not use explicitly modeled ontologies.

Always in the context of formal methods, other approaches use annotations with expressions that make explicit references to ontologies. Indeed, in [8, 71, 22], the authors argue that many problems in the development of correct systems could be better addressed through the separation of concerns. [8, 71] advocate the re-definition of design models correctness as a ternary relation linking the requirements, the system and application domains. Domain concepts are then explicitly modeled as first-class objects as we did in our approach. Furthermore, similarly to our approach, they propose the formalization of ontologies by Event-B contexts. The formalized information can then be integrated incrementally and directly in the behavioral requirements using refinements. In [22] a DSL abstract syntax and references to domain ontologies are axiomatized into logic theories. These two models are related using a third logical theory. The authors use the Alloy[57] formal method to check the consistency of the unified theory.

## 1.6 Ontologies and multi-view modeling

Several studies on the issue of multi-view modeling have been realized. Sirius [4], a part of EMF, proposes a multi-view approach which allows users to manipulate subparts of a model and focus on specific view of a design model. [63] presents the RAM approach, an aspect-oriented modeling approach that provides scalable multi-view modeling and however faces the global view consistency challenge. It focuses on the composition of UML class, state and sequence diagrams.

[35] presents an approach based software development with multiple viewpoints. Viewpoint templates are used to encapsulate the style (representation) and the workplan (specification method) corresponding to the different domain views of a system. A *viewpoint* framework, corresponding to the developed approach, and a logic-based approach to consistency handling are proposed later in [36].

[88] and [90] are the closest approaches to our work. [90] deals with the integration of viewpoints (points of view) in the development of mechatronic products. The vocabulary, assumptions and constraints of a specific domain are defined as *Viewpoint contracts* and *dependency models* (shared models) are used to capture the existing relations between the different views of a system. [88] proposes a framework for multi-view modeling to support Embedded systems Engineering. A common shared model, that consists of a combination of relevant knowledge from the domain-specific views, is

defined in SysML. The domain-specific views are directly generated from this common model which makes extensive use of SysML profiles and model transformations. [92] discuss the requirements for multi-view modeling and several approaches and tools are compared with regards to the defined requirements.

Compared to our approach, none of the work cited above, highlights the necessity of defining descriptive models and none of them makes use of explicit analysis models (points of view). Our approach improves these approaches. First, it defines explicit models that describes the whole features of an analysis. Second, it separates the descriptive domain information (ontologies) and the prescriptive system information (system's design model), it proposes a fully developed annotation methodology in order to strengthen system's design models by making explicit references to the domain knowledge. Both modularity and annotations insures that all the models we have defined (ontology, design model, point of view) can evolve asynchronously without impacting on the setted interactions with the other models.

## 1.7 Thesis outline

The outline of the rest of our dissertation is as follows. Next Chapter (Chapter 2) presents the defined general framework integrating our contributions. A general framework fulfilling the research goals ***RG1***, ***RG2*** and ***RG3*** is defined through the definition of a stepwise generic approach.

Two deployments of our general framework are proposed. Chapter 3 gives the details of the first deployment of our framework in a MDE setting. A stepwise model oriented approach is proposed. Then, meta-models are defined to support system's modeling, strengthening and multi-view analysis capabilities. Chapter 4 describes the second deployment of our general framework within the Event-B proof and refinement formal method. The involved models for design models strengthening and multi-analysis are formalized using set theory and predicate logic.

Chapter 5 describes the implementation of the approach in the context of a MDE setting. A Model Driven Engineering tool-chain supporting domain knowledge formalization, design models strengthening and multi-view analyses of design models is developed. The obtained prototype is provided as a set of plug-ins for Eclipse based on EMF[3] and Sirius[4].

We have validated our general framework on several non trivial case studies. Chapter 6 gives the particular details of the application of our framework on a Real-Time avionic system.

Finally, a conclusion closes our dissertation and proposes some perspectives on the future works.

# Part II

# Contributions

# Chapter 2

# General framework

Our proposed approach promotes strengthening complex systems design models and multi-analyses of these models by explicitly handling domain knowledge. This approach aims to define an engineering generic process that can be deployed in any setting to increase the quality of design models and multi-analyses by handling new hypotheses and properties entailed by making explicit the domain knowledge.

Thus, the main objective of this chapter is to reach ***RG1***, ***RG2*** and ***RG3*** by defining the general framework supporting strengthening and analysis of complex systems design models. Here, we give the definitions of the main steps and main artifacts involved in our approach. First, we discuss the formalization of domain knowledge within ontologies ***RG1***. Then we present in details the defined stepwise methodology to allow design models to explicitly handle domain knowledge formalized in domain ontologies ***RG2***. The stepwise methodology to handle explicit multi-analyses of systems is also described here ***RG3***.

Finally, this Chapter gives an introduction of the illustrative *Diplomas* case study we use along this thesis.

## 2.1 Handling domain knowledge in design and analysis of engineering models: global approach

Our global approach is composed of three main parts. The first one (RG1), deals with the explicit formalization of the domain knowledge, *domain ontologies* are dedicated for this purpose. This part is addressed in section 2.2.

The second part, represented on the left hand side of Figure 2.1 (RG2), addresses strengthening of design models. Design models are linked by references to domain ontologies that describe the knowledge associated to the concepts occurring in the

system model. A model based annotation methodology is developed for this purpose and its details are given in section 2.3.

The last part concerns the analysis of models through the definitions of *points of view* and *views*, it is depicted on the right hand side of Figure 2.1 (RG3). The model analysis methodology makes extensive use of the model annotation one. Indeed, a model annotation step is recommended in order to strengthen the quality of the system design model. Thus, the model analysis methodology will be triggered on the new enriched design model (output of the model annotation step) and the quality of the obtained analyses (*views*) will be increased. The model analysis approach is addressed in section 2.4.

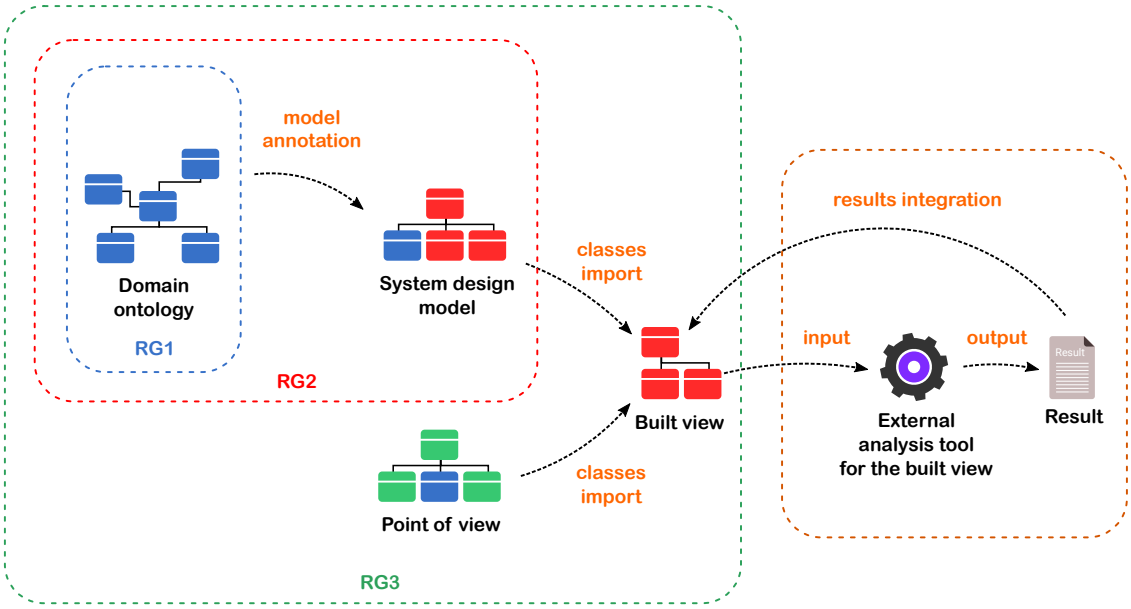



Figure 2.1: General framework for design and analysis of engineering models.

## 2.2 Ontologies formalization

The first step of our approach concerns the description of the domain knowledge. Information of the domain must be explicitly formalized before it can be integrated into design models and strengthen them. We use ontologies for this purpose Domain ontologies are, in fact, a simple and effective mean for representing the domain knowledge independently and in an asynchronous manner with the design models that are linked to it.

Ontologies formalization step plays a key role in the general framework process depicted in Figure 2.1. In fact, domain ontologies are required in order to strengthen system design models (gray box - Figure 2.1).

Different modeling languages may be used for building ontologies, design models and annotation models, leading to heterogeneous models. In order to integrate all models in a single setting, two solutions have been envisaged. The first one consists in using a single modeling language where all the models are described. The second one consists in using a single modeling language supporting meta-modeling capabilities. Then, each modeling language is described as a specific meta-model. We consider the second solution in our approach.

## 2.3 Strengthening design models using domain models: an annotation based approach

The second step of our approach concerns strengthening system design models with domain properties. We have defined a stepwise methodology in order to allow model designers to handle explicitly domain knowledge formalized in domain ontologies (section 2.2). The proposed approach consists in associating a domain ontology (concepts, properties and associated constraints) to entities of the design model. This association, is performed thanks to a defined annotation mechanism. Figure 2.2 highlights the annotation process depicted in Figure 2.1 and shows the overall schema of the model annotation approach. It involves four steps.

The defined annotation mechanism entails a loosely coupling of the ontology and of the annotated models. Indeed, no modification nor evolution of the design models is required. Moreover, ontologies and models may evolve asynchronously.

1. **Formalization of Domain Knowledge.** The information of the domain (concepts, links between these concepts, properties or these concepts and rules and constraints) are explicitly described and formalized in a knowledge model. Ontologies are used for this purpose and an ontology modeling language can be used to describe this model. The choice of this modeling language depends on the kind of reasoning to be performed. Note that this ontology shall be described independently of any context of use. It may also be built from already existing ontologies (e.g. standard ontologies).

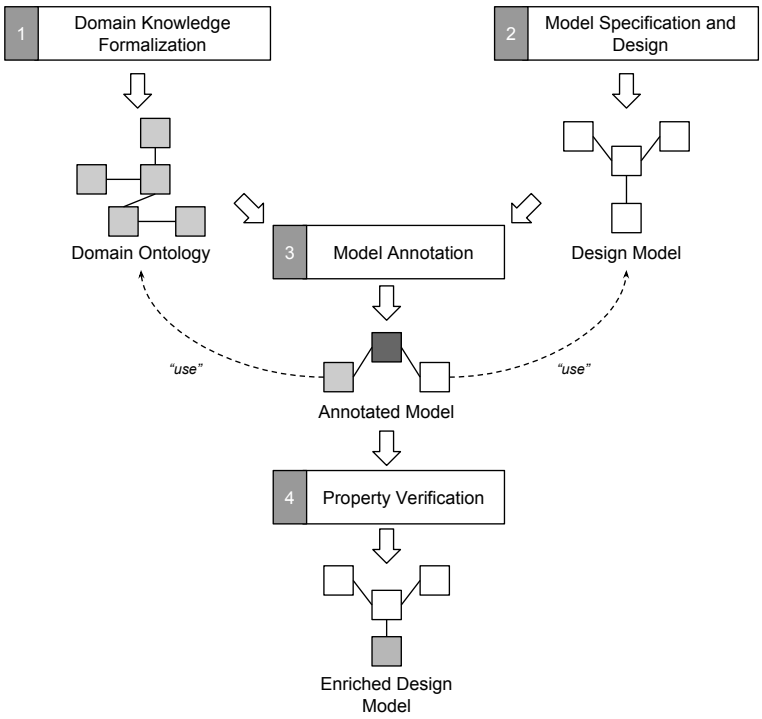



Figure 2.2: A four steps methodology for handling domain knowledge in models.

2. **Model Specification and Design.** Specific design models corresponding to a given specification are defined. They are formalized within a specific modeling language supporting different analyses, classically performed at the design modeling level.

3. **Model Annotation.** In this step, the relationships between design model entities and the corresponding domain knowledge concepts are identified. They correspond to model annotation. These relationships are themselves described with a modeling language. Different kinds of annotations relationships have been identified. Their explicit definition and details are given in the next chapters.

4. **Property Verification.** The annotated model obtained at the previous step is enriched by domain properties borrowed from the ontology. The annotated model is then analysed to determine whether, on the one hand, the properties and/or the constraints expressed on the annotated model are still valid and, on the other hand, new properties entailed by the annotation are valid.

At the end of this process, a new design model enriched with new domain information is obtained. Verification and validation of this model (step 4) are required

to check if the former properties and/or domain ones, resulting from annotation still hold.

### Some remarks

1. The languages used to model ontologies, design models and annotation relationships may differ. Semantic alignment between these modeling languages may be required. This topic is not addressed in our approach, we consider that these languages have the same ground semantics.

2. The engineering application we studied uses modeling languages with classical semantics using closed world assumption (CWA) [10].

3. It is important that the defined and used ontologies are related to the domain of the design model, and consensually defined by different stakeholders.

4. Steps 1 and 2 are independent. They may be performed in parallel. Ontologies may be predefined ones.

5. The definition of the annotations shall be realized either by manual, semi-automatic or automatic processes. However, only manual annotations are considered in our work and the guarantee of their soundness is left to the expert achieving the annotation.

6. Reasoning at ontology level shall be extendable as much as possible at the design model level thanks to the defined annotations mechanisms.

## 2.4 Multi-view modeling

In this section, we describe the work we have introduced in section 2.1 (right hand side of Figure 2.1) and achieved in order to make explicit the analysis of system design models.

A stepwise methodology to handle the multi-analyses of systems has been developed. The definition of this methodology comes from the observation of several experiments conducted by system developers. It is based on making explicit the knowledge related to the know-how associated to the performed analysis.

Figure 2.3 highlights the multi-view analysis process depicted in Figure 2.1 and details the defined methodology. This triptych describes what a system or model analysis is. We have identified three steps detailed in the following.

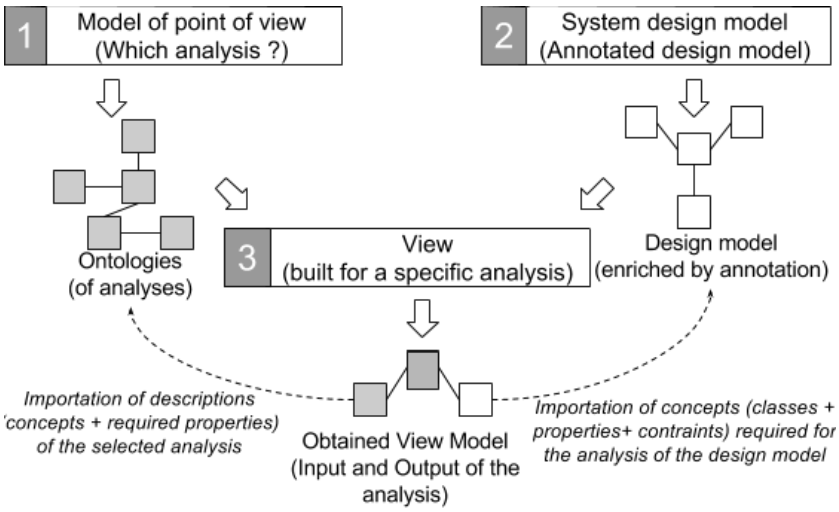



Figure 2.3: A generic approach for multi-view modeling.

1. **Model of point of view.** This step is related to the definition of a catalogue of system model analysis. It defines the notion of *point of view* which corresponds to the kind of analysis to be performed independently of any specific system or model. By catalogue, we mean an ontology describing all the relevant characteristics related to an analysis. This ontology shall mention all the required properties, the constraints, the algorithm and/or method used for a given analysis. An ontology modeling language is then required. It is defined as a meta-model in chapter 3 and formalized in an Event-B context using a deep modeling approach in chapter 4. It shall also organize these analyses with respect to the kind or type of analysis (following an ontological hierarchy classification based on the subsumption relationship - inheritance).

2. **System design model.** It consists in defining the model of the system to be analyzed. The choice of the right abstraction level is a key point. Indeed, if the chosen abstraction level leads to models that do not contain or represent the resources required by the analysis, then the chosen analysis cannot be performed. Refinement of the considered model is required to meet these requirements. Our approach is able to check this feasibility condition.

3. **View.** The integration of both analysis description obtained at Step 1 and the system model obtained at Step 2 is performed at the final Step 3. Here, the view corresponding to the definition of the *point of view* (obtained at Step 1) on the system model (obtained at Step 2) is built. Checking the availability

of all the information required by the analysis is performed at this level (i.e. checking the feasibility of the analysis).

We obtain at the end of this process a specific view model corresponding to the defined analysis. Instances of the view model are generated and the external tool in charge of the specific analysis (and described within the *point of view*) is triggered on this set of instances.

Finally, notice that although the above defined methodology relies on the definition of an integration of both *point of view* and system model, these two models are defined independently in an asynchronous manner. Second, our defined multi-view analysis methodology uses a single and shared modeling language for the description of the three involved models (*point of view*, design model and view model).

## 2.5 The *Diplomas* case study

In order to illustrate our proposal, we have chosen a didactic case study describing a simple information system. This information system results from requirements and is described through a set of concepts, actions and constraints as it is the case for applications in the engineering domain. The defined case study deals with the management of students diplomas and registration in the European higher education system. This system offers two kinds of curricula: first the Bachelor (Licence), Master and Phd , LMD for short, and second the Engineer curriculum.

Each diploma of the LMD curricula corresponds to a given level: Bachelor/Licence (high school degree + 6 semesters/180 ECTS credits), Master (Bachelor + 4 semesters/120 credits) and PhD (Master + 180 credits). Engineer curricula offers the engineer diploma five years after high school degree. Both Master and Engineer diplomas are obtained five years after high school degree.

### 2.5.1 Additional requirements for students registration

In the studied information system, students register to prepare their next expected diploma. This registration action takes into account the last hold academic degree (or last diploma) as a pre-requisite to register for the next diploma. Constraints on the registration action require that the information system does not allow a student to register for a new diploma if he/she does not have the necessary qualifications.

Therefore, the designed information system must check the logical sequence of obtained diplomas before allowing a student to register. For example, Phd degree registration is authorized only if the last obtained degree corresponds to a Master degree. The studied information system **prescribes** the necessary conditions for registering students for preparing diplomas.

Furthermore, the chosen case study is also concerned with the management of students diplomas. It offers, among other services, the printing of the diplomas of graduated students service (i.e. a specific view on the diploma information system). Some required information, carried out by the studied information system, are exploited to trigger this service.

### 2.5.2 Application of the general framework on the *Diplomas* case study

The deployment of our approach, presented in section 2.1, on the Diploma case study is described bellow.

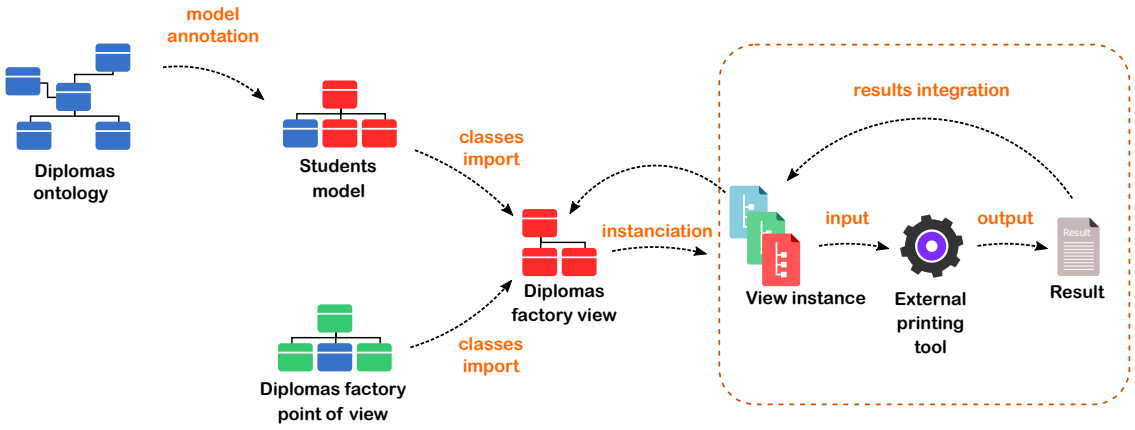



Figure 2.4: General workflow of the *student information system*.

Figure 2.4 depicts the overall schema of the analysis we have achieved on the Diploma case study. First, an annotation step is performed in order to enrich, strengthen and ensure the well definition (through the verification step - section 2.3) of the model. Then the *diploma factory* analysis is performed. The goal of the *diploma factory* analysis is to build a set of valid *printing instances* carrying all the necessary information required to trigger the external printing tool and to print student diplomas. These *printing instances* are directly extracted from the *student information system* instances.

We obtain at the end of the process a set of required instances (instances of the obtained *diploma factory* view) to trigger the external printing tool in charge of printing the students diplomas.

## 2.6 Conclusion

In this chapter, we have presented our general framework for models strengthening and multi-view analysis. Its deployment in the case Models driven engineering and formal methods one based on proof and refinement along with their illustration on the Diplomas case study is detailed through chapter 3 and 4.

First, ontologies are used to formalize domain knowledge and ontology modeling languages unifying their notations are proposed in both MDE and Formal setting. Second, our stepwise general approach for model strengthening has been detailed. Design models are enriched by making explicit references to domain ontologies. Annotation mechanisms are defined in both MDE and formal setting. They are powerful and allows a developer to map any entity of a design model to another one of an ontology without changing or modifying the original models. Third, the multi-view analyses approach has been presented. It defines explicit models that describe the whole features of an analysis. This approach makes an extensive use of the model strengthening one. Indeed, a model annotation step is recommended in order to strengthen the quality of the system design models to analyze. To the best of our knowledge, our approach is the only one that focuses on the importance of the explicit definition of a point of view.

We only considered, in this thesis, the case where the different involved models (ontologies, design models, analysis models) are described in the same modeling language thus, share a common ground semantics. The case of semantic mismatch where ontologies, design models and points of view are not described in the same modeling language should be considered as further extension of our work.

Our global proposed approach is based on the separation of the descriptive domain information (ontologies, points of view) and the prescriptive system information (system's design models). Hence, both modularity and annotations ensure that all the models we have defined (ontology, design model, point of view) can evolve asynchronously without impacting on the interactions set with the other models.

Finally, the work achieved in this thesis has been done as part of ANR-IMPEX and AME CORAC-PANDA projects. Prototypes corresponding to the deployment

of our approach in the MDE and Formal setting have been developed and presented in chapter 5.

The developed approach has been applied on several use cases.

# Chapter 3

# General framework: MDE setting

In this chapter, we show how the framework defined in chapter 2 can be deployed in a Model Driven Engineering (MDE) setting. We propose an incremental, model oriented approach, for strengthening design models and their multi-view analyses. Model-Driven Engineering (MDE) allows the modular representation of our solution and provides a very useful contribution for the design of trusted systems, since it bridges the gap between design issues and implementation concerns.

In what follows, we first discuss the ontology formalization step, an ontology modeling language is set for this purpose. The deployment of the model strengthening approach based of four steps is detailed. An annotation meta-model as well as different types of annotation mechanisms are set. Then multi-view analyses approach is deployed. The details of the integration of the information coming from both the design model and the point of view to build a specific view in an MDE setting are formalized in a specific meta-model.

Finally, the Diplomas case study is used for illustrative matter at each step of the presented MDE deployment.

## 3.1 Ontologies formalization

Ontologies can be expressed using different modeling languages. We propose, for the purposes of our study, and in order to unify these notations, a meta-model to describe ontologies (pivot). It is a generic meta-model independent of other ontology modeling languages. Its definition comes from a deep observation and detailed analysis of different ontology modeling languages.

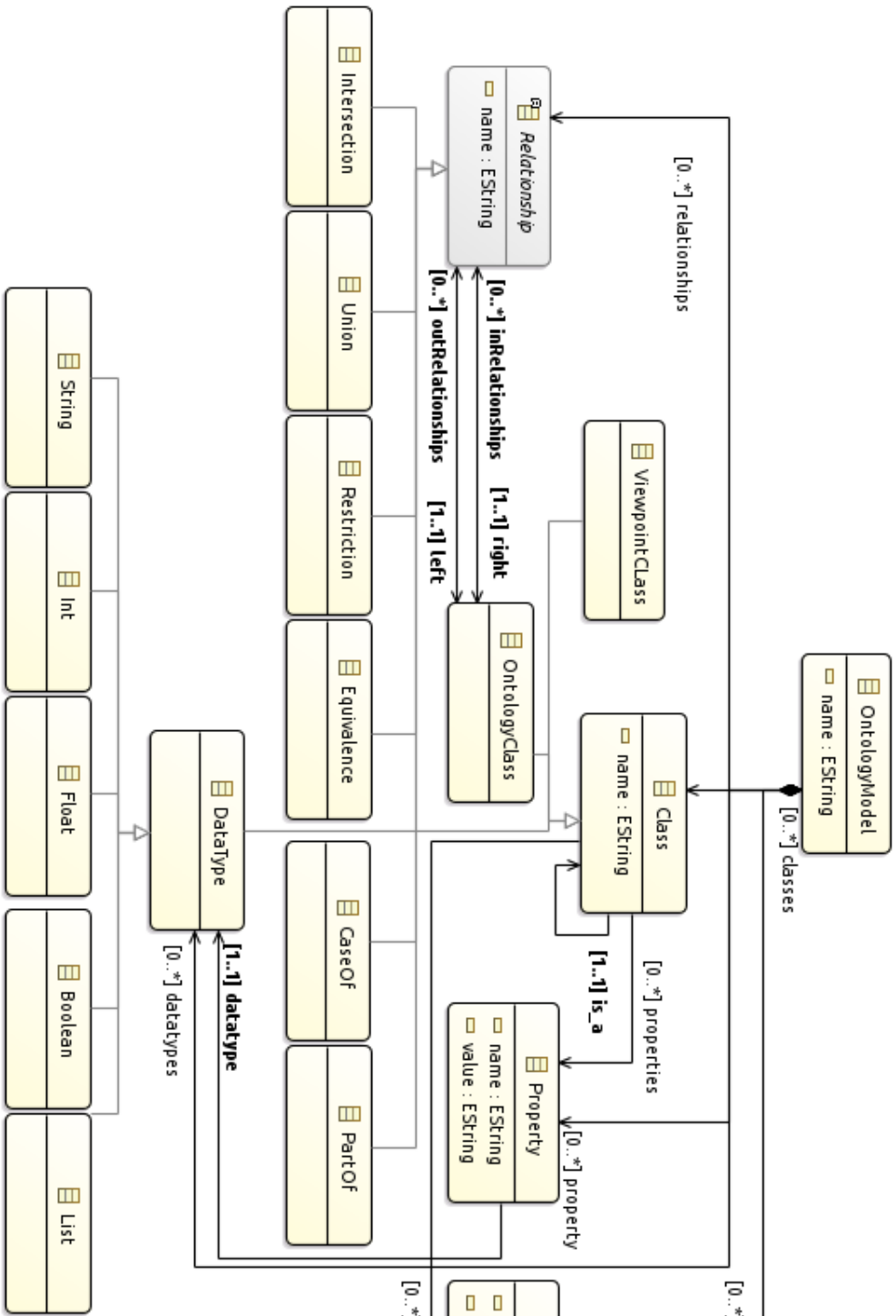



Figure 3.1: Ontology modeling language meta-model.

Thus, it gathers the main concepts available in various ontology modeling languages and can be extended with more ontology concepts. Moreover, this meta-model plays the role of a pivot ontology modeling language. It is shown in Figure 3.1 and its key concepts are described in the following.

- *OntologyModel* is the entry point of the model. An ontology model is composed of classes, datatypes, properties, relationships and constraints.

- *Class* represents an ontology class. A *Class* has properties, relationships, constraints and can be inherited from another *Class*. Different kinds of ontology concepts are defined as classes at meta level: Data type classes, point of view classes, view classes, etc.

- *Property* represent the properties of a *Class*. Note that each property has a type.

- *DataType* is a specific abstract *Class* referring to all the properties types. Constraints are defined at the meta-model level to restrict the *DataType* to their allowed value. For example, an *OntologyClass* can not contain *Viewpoint* properties.

- *String*, *Int*, *Float* and *Boolean* are all the basic types properties can be associated to.

- *Constraint* represent the classes or model constraints that can be formalized.

- The abstract concept *Relationship* represents the different relationships that may exist between the ontological concepts. A relationship links a source *Class* (left) to a target *Class* (right). A relationship may be of many types, we formalized four of them: *Equivalence*, *Restriction*, *Union*, *Intersection*.

The properties related to the semantics of the defined relationships are formalized as OCL constraints (e.g. properties related to *symmetry, reflexivity and transitivity* of the equivalence relationship). For example, Figure 3.2 gives the formalization of the transitivity property of the Equivalence relationship.

| | |
|---|---|
| $\forall$ x,y,z \| x$\mapsto$y $\in$ Eq $\wedge$ y$\mapsto$z $\in$ Eq $\Rightarrow$ x$\mapsto$z $\in$ Eq | `EQ_transitivity: Equivalence.allInstances()-> forAll(eq1, eq2\| eq1.right = eq2.left implies Equivalence.allInstances()-> exists(eq3\|eq2.right = eq3.right and eq1.left = eq3.left));` |

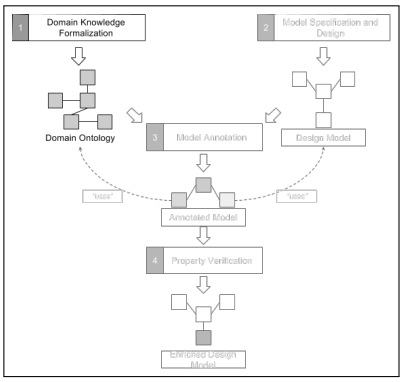
Figure 3.2: Equivalence relationship: Transitivity property expressed in OCL.

## 3.2 Strengthening design models using domain models: an annotation based approach

We show in this section how the proposed stepwise annotation methodology, presented in section 2.3 can be deployed in an MDE setting.

Figure 2.2 (presented in chapter 2 and recalled below for each step of the defined methodology) shows the overall schema of the approach involving four steps. Concepts, properties and constraints of the studied domain are represented and formalized within a knowledge model (domain ontologies) at step 1. Specific design models are defined at step 2. At step 3, relationships between design model entities and the corresponding knowledge concepts are identified. Three different kinds of relationships can be set up, they are discussed in section 3.2.3.2. Finally, at step 4, the annotated model is checked to determine whether the constraints associated to the knowledge domain, carried out by the annotations, can be expressed in the new enriched design model. The diplomas case study is used for illustrative purpose.

### 3.2.1 Step 1. Domain knowledge formalization




The deployment of our methodology requires, in its first step, the availability of an ontology formalizing the domain knowledge. The ontology is designed to integrate all the relevant properties of the domain, including its constraints.

Concepts and properties are modeled as classes and attributes of the ontology and the ontological constraints are added as OCL constraints. The obtained ontology model conforms to the ontology meta-model defined in section 3.1.

#### 3.2.1.1 An ontology for the *Diplomas* case study

The defined ontology for diplomas is depicted on Figure 3.3.

Diplomas and their characteristics represent a central knowledge for the previously defined *Diplomas* case study (but for other possible applications as well). A model to *describe* the diploma knowledge through diplomas characteristics, rules and constraints defines an ontology. It represents a shared knowledge model that can be used beyond the described application. The defined ontology contains a set of interrelated classes and relevant properties as follows.

- A subsumption relationship (represented by the *is_ a* relationship on Figure 3.3) is

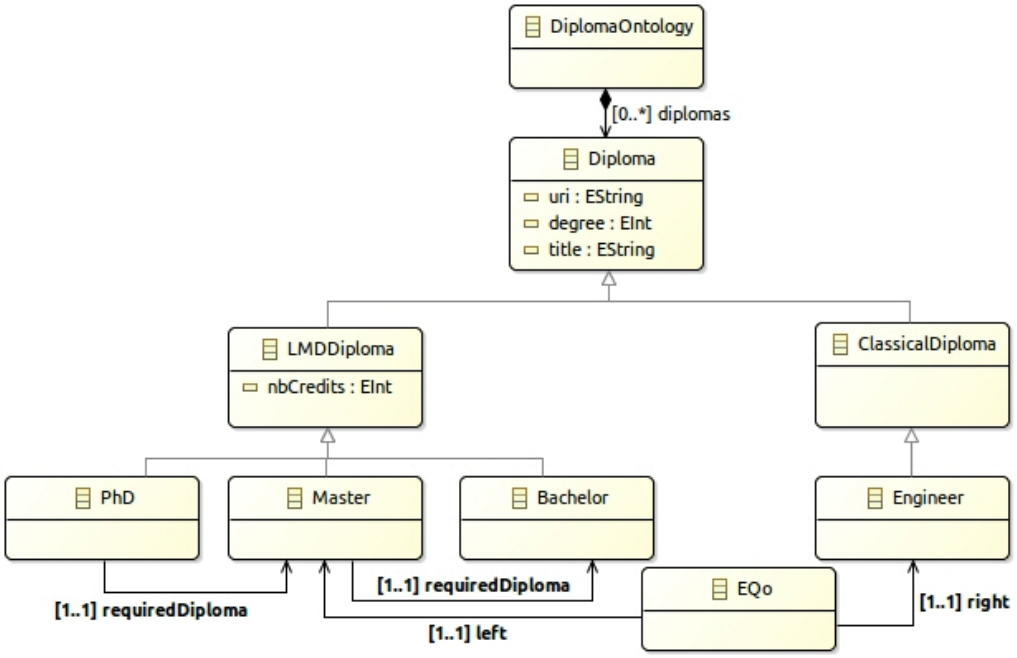



Figure 3.3: The *Diplomas* ontology.

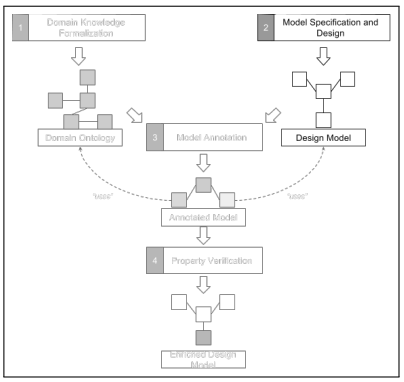
used to define hierarchies between categories of diplomas. *LMDDiploma* and *ClassicalDiploma* describe respectively the *Bachelor, Master* and *PhD* diplomas and other diplomas (e.g. *Engineer*).
- Several descriptive properties, like *title, degree, uri* of the *Diploma* class describe the name, the uri, the degree and the degree (level) of a given diploma. *nbCredit* defines the credit number required for each diploma.
- An ontological constraint on the model states that *Master* is equivalent to *Engineer*. It is written in the ontology modeling language as $EQ_o(Master, Engineer)$ where $EQ_o$ is an instance of the $Equivalent\_Class$ of the ontology meta-model.

In the ontology, this constraint is represented by an *equivalent* class linking the *left: Master* and *right: Engineer* classes of the same ontology. Another constraint defined as *thesisRequirement* carried by the *requiredDiploma* relationship ($requiredDiplom_i$ property) is added to assert that any master (or any equivalent diploma) is required to prepare a PhD.

Note that the presented Diplomas ontology is only one of the possible ontologies for describing the diplomas domain knowledge. A final domain ontology needs to be consensually defined.

### 3.2.2 Step 2. Model specification and design

The design models are defined by the designer according to a given specification. Several design models corresponding to particular designs for a problem requirement may be produced (these models may correspond to descriptive models, structural models or behavioral models). The designed models include specific design constraints expressed using a constraints language.




#### 3.2.2.1 A design model for the *Diploma* case study

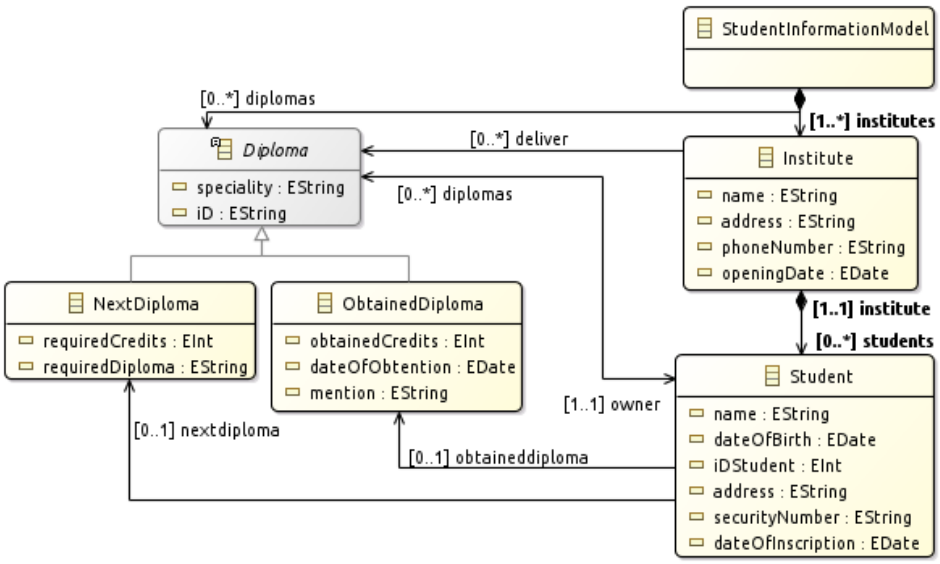



Figure 3.4: Engineering student model.

Figure 3.4 shows one possible UML class diagram representing the information system related to the management of students and their diplomas. It is composed of institutes and diplomas as follows.

- An institute (a university or an engineering school) is composed of its students to whom it delivers diplomas.

- A student is represented by *Student* class with the *name*, *dateOfBirth*, *iDStudent* (for student number), *address*, *securityNumber* (for social security number) and *dateOfRegistration* properties.

| | |
|---|---|
| Student.nextDiploma.iD = "p" ⇒ Student.obtainedDiploma.iD = "m" | `phdInscritpion: self.NextDiploma.iD = 'p' implies self.obtainedDiploma.iD = 'm'` |

Figure 3.5: Formalization of *phdInscritpion* constraint.

- A student is related to his/her *institute* and his/her *Diplomas*.

- A *Student* holds an *ObtainedDiploma* representing the last obtained diploma (*obtainedDiploma* relationship) and a *NextDiploma* referring to the next diploma in preparation (*nextDiploma* relationship).

- An institute is represented by the *Institute* Class with properties *name*, *address*, *phoneNumber* and *openingDate*.

- The *ObtainedDiploma* is characterized by the *dateOfObtention* (for date when the diploma was obtained by a student) and the *obtainedCredits* (for the number of credits a student obtained for his last diploma) properties.

- *nextDiploma* is characterised by the *requiredCredits* and *requiredDiploma* (for the number of credits and the grade of diploma required in order to register for a specific next diploma) properties.

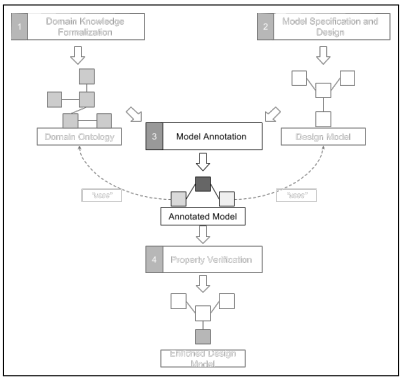
Finally, a constraint named *phdInscription* on the student *nextDiploma* is defined. It asserts that a student registering for a PhD diploma needs to hold a master diploma to be allowed to register for a PhD. It represents a model invariant and it is defined by the OCL constraint of Figure 3.5.

Observe that, even though this design model uses concepts (classes, properties, data types, etc.) that are semantically close to those of the ontology defined in section 3.2.1.1, no explicit reference nor link to this ontology is offered.

### 3.2.3 Step 3. Model annotation

In step 3, relations are set up between the design model entities and the ontology concepts. The annotation relationships are explicitly defined in an annotation model to keep trace of the annotation process and to ensure asynchronous evolution of both the design model and the ontology.

### 3.2.3.1 Core classes for model annotation




The relevant information and entities required to set up the methodology depicted on Figure 2.2 (section 2.3) are summarized in a simplified class diagram on Figure 3.6.

In step 3 of the model annotation approach relations defining the annotation model are established between the design model entities and the ontology concepts. These annotation relationships link between design model entities (classes, properties, datatypes, associations, etc. ) and ontology concepts (classes, properties, associations, etc.).

Figure 3.6 depicts an extract of the annotation meta-model where the annotation relationships are formalized. This meta-model is defined as an extension of the meta-models of both the ontology and design models. It integrates the annotation mechanism at a meta-modeling level.

The annotation class *ClassAnnotation* is defined to link (annotate) a design model class (ex. *ModelClass*) with an explicit reference to an ontology concept (ex. *OntologyClass*). Other types of annotation classes, like *InstanceAnnotation* and *PropertyAnnotation*, etc. are also defined. They are used to annotate other entities of the design model (instances, properties, etc.).

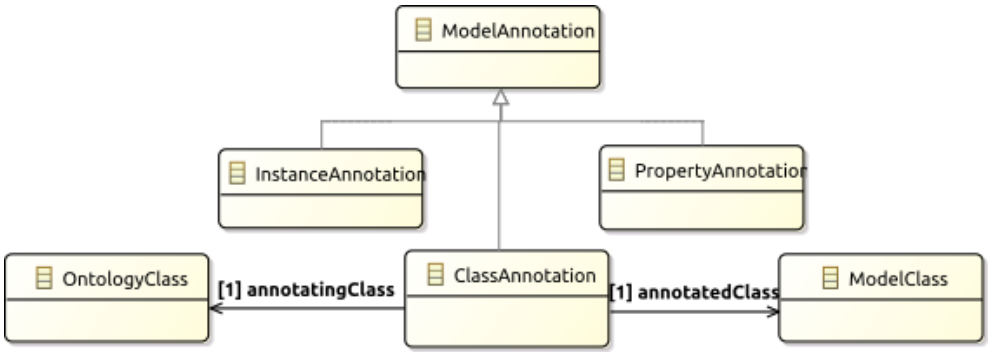
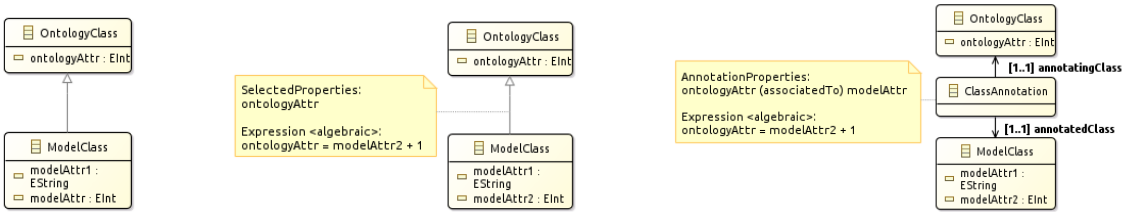



Figure 3.6: Core classes for model annotation.

The annotation step described above requires the definition of specific annotation mechanisms. Different kinds of annotation mechanisms have been set up: inheritance, partial inheritance and algebraic relationships [44, 45]. Other annotation mechanisms can be defined for specific cases if required. The details and choice of the right mechanism are given in the next subsection.




(a) Annotation by inheritance

(b) Annotation by partial inheritance

(c) Annotation by association

Figure 3.7: Annotations mechanisms

#### 3.2.3.2 Model annotation: three identified cases

At step 3, model annotations formalized as relations, are established between design model entities and ontology concepts. We have identified three annotation mechanisms[42] which link design model entities (classes, properties, datatypes, associations, etc. ) and ontology concepts (classes, properties, associations, etc.).

**Annotation by inheritance** is defined by the *Is_ a* relationship (subsumption relationship [59]) . In this case, a concept of the ontology subsumes an entity of the design model. The mapping relationship is the subsumption (is_a) relationship. All properties, attributes, rules and constraints that apply to the ontological concept become also applicable to the design model entity.
The annotation by inheritance maintains the ontological reasoning and preserves it at the design model level. But, note that due to the inheritance of all the resources issued from an ontological concept, all these resources are expressible at the model level. However, it may happen that some of these properties are not valuable at instance level (after annotation). This relationship is usually set up in an a priori setting where the ontology is designed before the design models are defined. Figure 3.7(a) depicts an illustration of the annotation by inheritance where *ModelClass* is subsumed by (inherited from) *OntologyClass*. Thus, the *attribute1* ontological property becomes directly available in *ModelClass* thanks to the inheritance mechanism.

**Annotation by partial inheritance** is defined by the *Is_ case_ of* relationship. It is also a subsumption relationship. It defines a partial inheritance relationship [59]. This relation behaves like the Is_a relationship, except that it does not require the inheritance of all the ontological class properties. In fact, only some

of the relevant properties and constraints of the ontology class are imported (inherited). The annotation mechanism is in charge of selecting which properties and constraints are imported. Here again, some of the domain restrictions (constraints) formalized in the ontological classes participating to the annotation may not be expressible at design model level if the properties they are related to are not valuables within this model.
The main advantage of this approach is flexibility, it can be set up in any situation (a priori and a posteriori). An illustration of the annotation by partial inheritance is depicted in Figure 3.7(b) where *ModelClass* is annotated by *OntologyClass* using partial inheritance mechanism. *ontologyAttr* ontological property is selected to be inherited in *ModelClass* and an *algebraic expression* makes explicit the existing correspondence between *ontologyAttr* ontological property and *modelAttr2* model property.

**Annotation by association** *Is_ a* and *Is_ case_ of* relationships are based on relationships available in the ontology model. It may happen that an annotation needs specific relationships defined by the users to define specific mappings. These relationships are themselves described in ontologies. This annotation enables the connection of ontological classes with model classes by association. In this case, subsumption reasoning contained the ontology is not preserved at the annotated design model. But, the properties borrowed from the association to the design model can be used to express model properties. Figure 3.7(c) depicts a generic illustration of an annotation by association. *ModelClass* is annotated by *OntologyClass* using the association mechanism. *ontologyAttr* ontological property is associated to *modelAttr2* model property and an *algebraic expression* defines the exact correspondence between these two properties (*ontologyAttr = modelAttr2 + 1*).

#### 3.2.3.3 The *Diploma* case study annotation

Figure 3.8 shows how the annotation relationships between the students design model entities and the Diplomas ontology concepts are set up. The annotation by association mechanism is used for this purpose.

The *ObtainedDiploma* class of the students design model instantiates *ModelClass* (Figure 3.6) and the *Master* class of the diplomas ontology instantiates *OntologylClass* (Figure 3.6). The *ObtainedDiploma* class is annotated by making explicit references to the Master class using a *ClassAnnotation* class. Similarly, *NextDiploma* is annotated

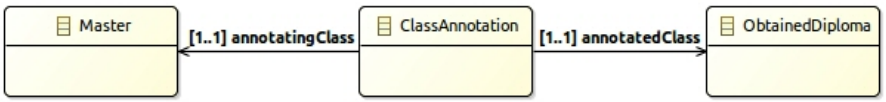
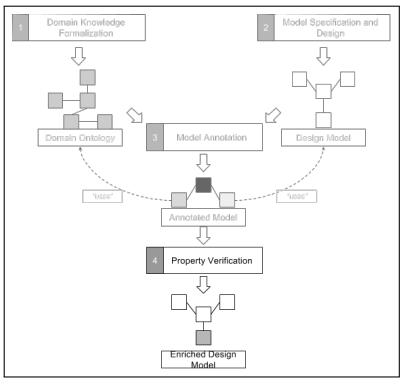



Figure 3.8: Annotation of Student model.

by *PhD* of the Diplomas ontology. The non-structural property *Equivalent_ Class* and the *thesisRequirement* constraint can now be accessed and exploited. So, the equivalence between Master and Engineer classes is expressed and made explicit within the design model.

### 3.2.4 Step 4. Properties verification

The last step of the approach analyses the obtained annotated design models through formally established links with the ontology. The defined annotation process leads to the enrichment of the original design model with new relations, properties, constraints and rules. Ontological properties and classes are considered to be available in the enriched model if they have been explicitly selected or linked to model properties during the annotation process (third step of the approach).




It may happen that these relations, properties and constraints could not be expressed at the design model level and thus not evaluable at instantiation level due to the absence of attributes to express them and of the values of these attributes (instances). These constraints become meaningless. At this level, an analysis of the obtained relations, properties, constraints and rules issued from the annotation is necessary after an annotation by *Is_ Case_ Of* or by *association* because these two types of annotation offer the possibility of having only certain ontological properties in the enriched design model. The annotation by *Is_ a* does not suffer from this drawback since all ontological constraints in the design model can be expressed (all the properties of the annotating ontological classes are inherited in the design model). The proposed analysis procedure is depicted on figure 3.9. It illustrates the execution of the verification step using an algorithm.

The process begins by selecting an annotated class in the design model and analyzes it to retrieve the ontological class that annotates it. Each constraint, property

```
BEGIN

   For (an annotated model)
   begin
        Select a new annotated class;
        Select the corresponding ontology class;
        For (all ontology class constraints)
        begin
           Select a new constraint;
           if (constraint is expressible in the domain model) then
              integrate to the domain model;
           else
              add an error message;
        end;
    end;
END;
```

Figure 3.9: Algorithm of the verification process

and relation of the annotating class is then checked to decide of its expressiveness in the design model. The expressible entities are integrated into the model, the other ones are returned to the user for information purpose.

The new enriched model is then finally validated by (re-)checking all the constraints (the existing and the new added ones) on the model and all its instances.

#### 3.2.4.1 The *Diplomas* case study verification

The *equivalence* property and the *thesisRequirement* constraint are now explicit on the annotated student model.

The verification process ends with integrating the *equivalence* domain constraint into the enriched design model since all the properties it is related to are available. At this level, it becomes possible to conclude that a student can apply for preparing a Phd thesis if he holds an engineer diploma. Thus, the *phdInscritpion* constraint depicted in Figure 3.5 is rewritten (manually) and its formalization is given in Figure 3.10. The new *phdInscritpion* property integrates the result of the set up annotation (blue color) and thus, became explicit after handling domain knowledge (by annotation) expressed in the ontology.

At the end, the obtained model together with its instances are now ready to be analysed. Figure 3.11 shows an example of instance for the enriched student information system model. Instances of *Student*, *School*, *ObtainedDiploma* and *NextDiploma* classes (bullet 1 of Figure 3.11) with all their associated attributes are defined. For

| | |
|---|---|
| Student.nextDiploma.iD="P"<br>⇒<br>annotation(Student.obtainedDiploma)<br>∈ eq(Master) | phdInscritpion: self.nextDiploma.iD = 'p' implies<br>let c: ecore::EClass = ClassAnnotation.allInstances()-<br>select(inst\|inst.annotatedClass =<br>self.obtainedDiploma)- at(0).annotatingClass<br>in Equivalence.allInstances()- exists(eq\|<br>eq.left.uri = 'Master_uri' and eq.right = c); |

Figure 3.10: The OCL constraint *phdInscritpion* after annotation.

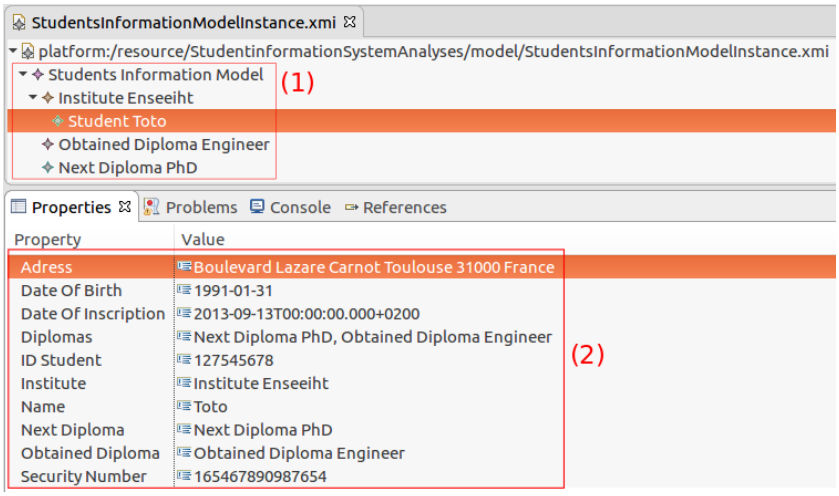



Figure 3.11: *Student information system* model instance.

example, the values of *Student* instance properties are shown in bullet 2 of Figure 3.11.

Next section shows how the diplomas factory analysis is performed on these model and its instances.

## 3.3 Multi-view modeling

We show below how the proposed stepwise multi-view modeling methodology can be deployed in an MDE setting. The Diplomas case study is used for illustrative purpose.

### 3.3.1 The core model elements

The relevant information and concepts required to set up the methodology depicted on Figure 2.3 are summarized in a simplified class diagram on Figure 3.12. The following relevant properties are required in order to obtain the integrated view corresponding to a system model analysis. Their details are given in the next subsections.

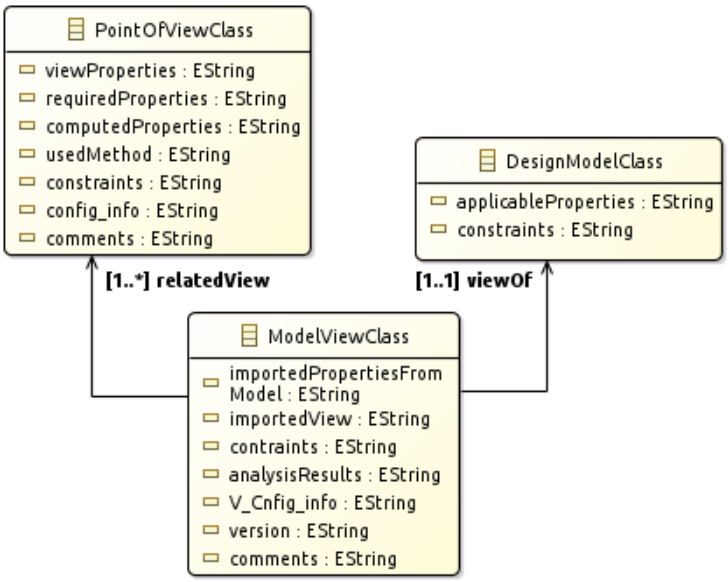
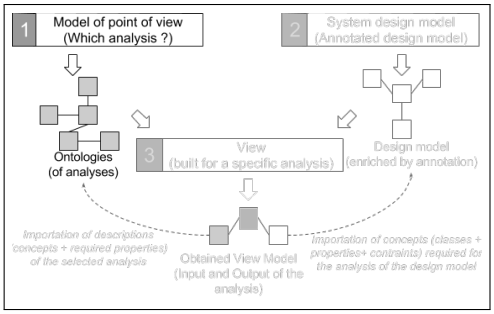



Figure 3.12: Core classes for multi-view modeling

#### 3.3.1.1 Step 1. Model of point of view definition




The deployment of our model analysis methodology requires, in its first step, the description of a point of view. To make this description explicit, we have used a simple class diagram (Figure 3.12) to express the different properties required to perform the analysis.

The *PointOfViewClass* of figure 3.12 corresponds to the description of an analysis defined at step 1 of Figure 2.3. The following properties are defined.

- The *viewProperties* property is associated to the view. It characterizes the descriptive and specific properties of an analysis. Thus, it gathers all the additional information. For example, it allows keeping trace of the view details by adding information to view like the name of the analysis author (ie. analysis expert), view version, analysis launching date, etc.

- The *requiredProperties* property defines the set of properties of the system model needed in order to trigger the described analysis. Mappings (exploiting the previously defined annotation methodology) may be required to map the properties defined in the point of view with those defined in the system model.

- The *computedProperties* property describes the output of the analysis corresponding to the currently described point of view or analysis.

- The *usedMethod* property defines the specific technique, method or program that supports the defined analysis. It may be characterized by a function taking the *requiredProperties* as input and returning *computedProperties* as output.

- The *constraints* property defines the constraints imposed by the method to be executed. It concerns constraints related to space or processor or any other required hypotheses.

**A point of view for the *Diplomas* case study.** The defined *Diplomas factory* point of view for printing students diplomas from the *Student information system* is depicted in Figure 3.13. The external *printing function* to be triggered and its required input parameters are described. The *printingMethod* property characterises the external *printing function* to be used and the *printingTool* property makes references to the external tool (encoding the printing function) that will be called for printing students diplomas.

Point of view's (PoV) *requiredproperties* and *viewproperties* make references to the needed input parameters of the diploma *printing function*: *name* of a student, *dateOfObtention* of a diploma, *iDStudent* (for student identification number), etc. are described as required properties. Thus, they shall be imported directly from the design model. The *paperSize* and the *logo* of the university are defined as view properties and are directly imported from the corresponding ontologies. The *Result* class defining the *url* property is used to store the output results of the *printing function*. The model of Figure 3.13 is obtained.

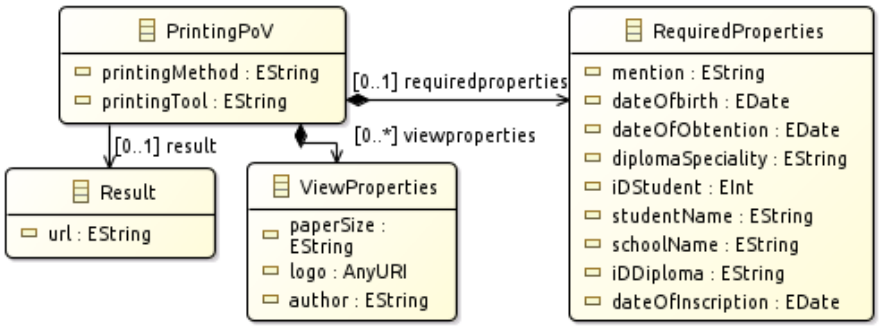
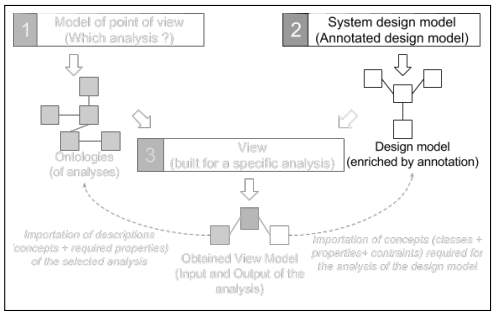



Figure 3.13: The *diplomas factory* point of view.

Notice that the semantics of these properties (required properties and view ones) are defined in the corresponding domain ontologies (diplomas ontology, students ontology, printing ontology, etc.). Moreover, not all the design model properties are required for the construction of a *Diplomas factory* view, thus only the required properties, specified within the point of view model, are imported for each specific analysis.

#### 3.3.1.2 Step 2. System design model definition

*DesignModelClass* of Figure 3.12 corresponds to the information model describeing the models to be analyzed (step 2 of Figure 2.3). It is formalized within a modeling language that supports different analyses. We may find at least what follows.



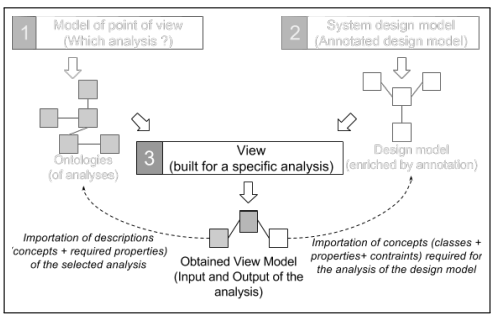

- The *applicableProperties* property corresponds to all properties associated to the classes of the design model to be analyzed. Thus, the required properties (*requiredProperties* of *PointOfViewClass*) for building a specific view are selected from these model properties.

- The *constraints* property corresponds to the domain constraints that are defined on a design model. These constraints characterize the correct construction of the design model to be analyzed and its set of instances. They may be imported into a specific view to preserve and guarantee the correctness of the obtained view.

**A design model for the *Diplomas* case study.** The design model supporting multi-analyses is defined at this level. It corresponds, in this case study, to the new strengthened *student information system* model and its instances obtained at the end of the *model annotation* step (subsection 3.2.4.1).

#### 3.3.1.3 Step 3. Building the view

Finally, at step 3 of the approach, the integrated view is built by composing the resources issued from both concepts of step 1 and step 2. The defined view is characterized in Figure



3.12 by the *ViewModelClass* class and its corresponding properties as follows.

- The *importedPropertiesFromModel* property corresponds to the set of properties imported from the design model to be analyzed. It defines the properties needed from the model (*requiredProperties*) to build the view and perform the analysis.

- The *importedView* property refers to the description of analysis to be performed on the considered model. Information about the selected point of view, name of the analysis, the used method, the tool to be triggered, etc. are imported into the view to make explicit and keep trace of all the choices made by the analysis expert in order to build the specific view.

- The *constraints* property defines the new constraints (*PointOfViewClass* constraints and *DesignModelClass* ones) that apply on the integrated view once imported in it. They guarantee the correct construction of the the view model and its corresponding instances.

- The *analysisResult* property defines the property containing the results of the analysis. In fact once an analysis is completed, the corresponding result is made explicit and integrated into the view model using this property. The result can then be interpreted treated by the analysis expert.

**Building the *Diplomas* factory view.** A specific *Diplomas factory* view is built by integrating the resources issued from both the *student information system* model and the *Diplomas factory* point of view. Thus, both the *required properties* and the *view properties* are imported in the *Diplomas factory* view. Figure 3.14 depicts the resulting view.

- *PrintingPoV*, *Viewproperties* and *Result* classes are directly imported from the *Diplomas factory* point of view. They contain the specific properties *Diploma factory* view.

- *Student*, *ObtainedDiploma*, *Institute* classes are directly imported from the design model (Figure 3.4). They contain all (and only) the required properties referenced by the *Diplomas factory* point of view (Figure 3.13).

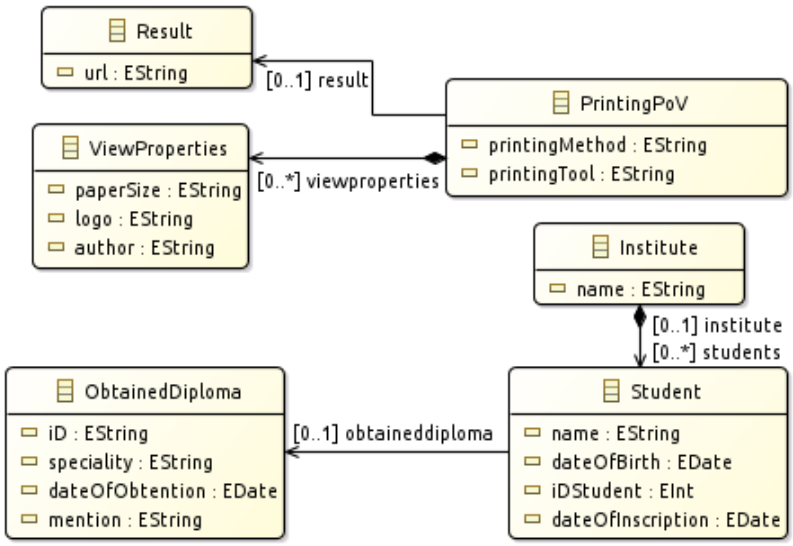



Figure 3.14: Diplomas factory view.

The instances corresponding to these classes and properties are generated to build the *Diplomas factory* view instances. Figure 3.15 depicts such a *Diplomas factory* view instance. It is built (extracted) from the enriched *student information system* model instance shown in Figure 3.11. Only the relevant information for *diplomas factory* view are defined.

- Instances of *ObtainedDiploma*(bullet 1 of Figure 3.15), *Institute* (bullet 3 of Figure 3.15) and *Student* (bullet 4 of Figure 3.15) classes containing -only- the the values of the *required properties* are imported. For example, *Date Of Birth*, *Date Of Inscription*, *ID Student*, *Institute*, *Name* and *Obtaineddiploma* properties values of the *Student* class instance (depicted in bullet 7 of Figure 3.15) are imported into the view instance since they explicitly described as required properties from the model in the *Diploma printing* point of view depicted in Figure 3.13.

- The *usedMethod*, *usedTool* properties are valued within the *PrintingPoV* class instance (bullet 5) to describe the suited analysis to be triggered on the view instances.

- The view properties like *paperSize*, *logo* and *author* are also valued within the *View Properties* class instance (bullet 6). The additional user choices are made explicit.

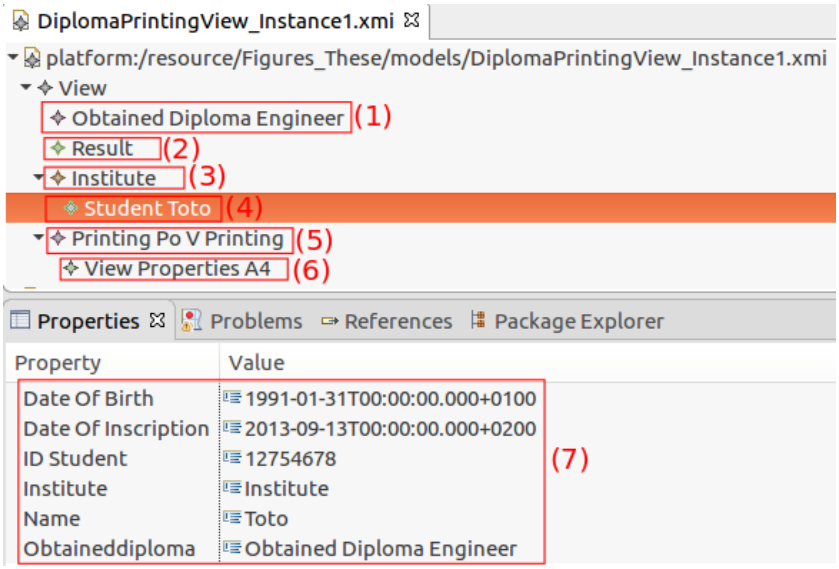



Figure 3.15: *Diplomas factory* view instance.

- An instance of the *Result* class is generated for each instance of the Diplomas factory view, it contains the *url* (address) of the output of the triggered external printing tool. In this case the location of each printed diploma can be accessed.

- Finally, instances of *NextDiploma* class are not imported in the view instance since they are not relevant for this view (not described within the *Diplomas factory* point of view).

At the end, the external printing tool is triggered on the set of *Diplomas factory* view instances. Students diplomas are printed and their corresponding storing *url* are given as output results of the printing tool.

### Some remarks

The previous resources represent the concepts of a meta-model describing the integration of a point of view and a design model in order to obtain a specific view. Note that the list of the given properties is not exhaustive, other properties to describe configuration information, analysis expert comments, etc. can be added.

The presented meta-model (Figure 3.12) also defines the constraints that guarantee the correct integration of a point of view and a design model. These constraints define the correct correspondence between *applicableProperties* of the design model and the *requiredProperties* of the point of view. They are checked in order to guarantee the correct construction of the view. Thus, a construction is considered correct only if all the required properties (*requiredProperties*) can be retrieved within the

defined design model properties (*applicableProperties*). These properties correspondences are made possible through the explicit references to ontologies. In fact, if two properties refer to the same *uri* of an ontological property, then they are considered to be semantically equivalent (identical).

## 3.4 Conclusion

In this chapter, we have presented the Model Driven Engineering setting of our general framework for handling domain knowledge in design and analysis of design Models. First, we have defined an ontology modeling language in order to be able to express ontologies formalizing domain knowledge. The proposed ontology modeling language has been set by analyzing different existing modeling languages and can be extended. Second, we have presented the MDE setting for strengthening design models. An annotation meta-model has been defined and three annotation mechanisms have been set: annotation by *inheritance*, *partial inheritance* and by *association*. Third, we have given the details of the multi-view analysis approach. The core model elements linking a design model and a point of view in order to extract a specific view have been described.

Notice that the models involved in our approach (ontology, point of view, annotation model) are associated with domain constraints (or algebraic constraints for properties annotation in case of annotation *by partial inheritance* and *by association*). Thus, the defined meta-models for ontology formalization, model annotation and multi-view analysis should be extended to define and integrate a language for constraints description together with a procedure to solve and verify them.

A prototype built on the EMF Eclipse platform has been produced and its details are given in chapter 5. Moreover, this approach has been applied on different case studies among which the *avionic embedded systems* case study presented in chapter 6.

Finally, the presented MDE setting does not offer any formal proof context to guarantee the correctness of all the models involved in the presented approach. Thus, this is done by defining domain constraints expressed in a constraints description language (for instance OCL) and checked on all and each instance of these models.

Next, we present the Event-B formal method setting based on proof and refinement for handling domain knowledge in design and analysis of system Models.

# Chapter 4

# General framework: Event-B formal method setting

According to the stepwise methodology defined in the general framework (chapter 2 for domain knowledge handling in design and multi-analysis of models, we propose in this chapter a general formal setting in which such a methodology can be deployed for specific formal methods. The involved models are correct by construction. The Event-B proof and refinement formal method is used. The models are formalized using axioms and strengthened with proved theorems that are directly deduced from these axioms.

First, a generic Event-B context for ontologies is proposed. It formalizes the main ontology relationships and thus, is extended to formalize specific domain ontologies. The model strengthening approach is set and a generic annotation relationship is defined in a generic annotation Event-B context. The multi-view analysis methodology is also deployed in the Formal Event-B setting. Points of view are described in Event-B contexts and views are formalized within Event-B machines.

As for the MDE deployment, the Diplomas case study is used for illustrative matter.

## 4.1 Ontologies formalization

In this section, we give the details of the formalization of the ontology language in formal setting.

We have identified that specific modeling languages support the definition of ontologies. The availability of such languages means that specific semantic constructs are associated to these languages. Therefore, the need to explicitly represent such semantic constructs when reasoning formally on ontologies appears. One way to

model such ontology constructs is to use the constructs offered by Event-B language to explicitly model the ontology modeling language constructs.

In the context of the IMPEX project, we have identified two mechanisms to define ontologies as formal theories. These two approaches use two different modeling approaches: shallow and deep modeling.

### 4.1.1 Shallow modeling

The shallow modeling approach consists in formalizing the ontology concepts directly in the target modeling language without keeping trace of the structure of the ontology modeling language concepts [72].

This formal modeling of ontologies is based on existing ontologies models expressed into languages such as OWL, which are formalized as an Event-B specification (context) by means of transformations rules. The transformation rules describe the corresponding sets, constants, axioms to each ontological source construct. More detail about shallow modeling of ontologies developed in the context of IMPEX Project can be found in [72].

Listing 4.1 presents an extract of the diplomas ontology formalized using the developed transformation rules into an Event-B context. It illustrates the use of the subsumption (*Is_a*) and *equivalence* relationships as follows.

Listing 4.1: Formalization of the Diplomas ontology using Shallow modeling.

```
Context Diplomas_Ontology

Sets Thing

Constants Phd, Master, Engineer, Diploms, Bachelor

Axioms
  axm1: Diploms ⊆ Thing
  axm2: Phd ⊆ Diploms
  axm3: Master ⊆ Diploms
  axm4: Engineer ⊆ Diploms
  axm5: Bachelor ⊆ Diploms
  axm6: Engineer = Master
  ...

End
```

- A global set *Thing* is defined, it refers to the root ontology class *Thing*;
- *Diploms* is defined as a subset of *Thing*;
- *Phd*, *Master*, *Engineer* and *Bachelor* ontology classes are formalized as subsets of *Diploms* (*axm1*, *axm2*, *axm3* and *axm4*).

- The equivalence relationship between *Engineer* and *Master* diplomas is defined by means of an equality between the sets *Engineer* and *Master* in *axm6*.

Using shallow modeling, the information related to the modeling language are not explicitly defined in the target formal modeling language. Reasoning on ontologies is performed using the underlying proof system of Event-B. Next, we present the deep modeling deployment of domain ontologies which allows the explicit formalization of ontological entities and its semantic constructs.

### 4.1.2 Deep modeling: ontology language formalization within a context

Deep modeling considers explicit modeling of both ontology modeling concepts and ontologies. Thus, ontologies are defined as instances of ontology models. The interest of this approach is that the ontology model is explicitly described and reasoning on ontologies is performed using the proof system described in the ontology model and the underlying proof system of Event-B.

To model ontologies in deep modeling, two steps are required. First, an ontology model is formalized and then ontologies are defined as specific models (instances) corresponding to the defined ontology model.

The deployment of our approach in a formal method setting is based on deep modeling.

### 4.1.3 Our ontologies formalization: deep modeling

In the following, we show how semantic ontological constructs (types, relationships, etc.) are modeled as Event-B contexts and then how specific ontologies can be described using the Diplomas case study. We also show how reasoning rules available in ontology reasoners can be formally expressed and proved.

According to the characteristics of the modeling languages reported in [54], we have described a formal Event-B context for canonical concepts and non-canonical ones. We have collected the main constructs available in the OWL[79] and Plib[55]. Their semantics is specified within the event-B axioms associated to each ontological construct.

#### 4.1.3.1 Canonical concepts

Canonical concepts represent the core entities of an ontology. Among these concepts, we find classes, properties, types and instances. The subsumption relationship offers the capability to define a hierarchy of concepts.

**Formalization of basic concepts.** The basic resources are described in an Event-B context. Listing 4.2 gives an extract of the *Ontology_Relations* Event-B context. It defines through relations the whole basic and canonical concepts available in an ontology modeling language. Sets represent the unique carrier to describe such concepts. The set of axioms of the $AXIOMS$ clause is completed by the relevant properties, definitions associated to the defined sets and relations. The concepts are described as follows.

Listing 4.2: Canonical concepts of an ontology language.

```
Context Ontology_Relations

Sets
   CLASS,
   PROPERTY,
   INSTANCE,
   VALUES,
   ...

Axioms
   axm1: HAS_PROPERTIES = CLASS ↔ PROPERTY

   axm2: HAS_INSTANCES = CLASS ↔ INSTANCE

   axm3: HAS_VALUES = INSTANCE × PROPERTY ↔ VALUES

    ...
```

- $CLASS$, $PROPERTY$, $INSTANCE$ and $VALUE$ carrier sets define classes, properties, instances and values entities of an ontology. These sets are abstractly defined, they will be populated when defining specific ontologies.

- $HAS\_PROPERTIES$ relationship is defined to associate properties to classes and explicitly set a class properties. This is achieved using the Event-B relation operator $\leftrightarrow$. This notation is used to combine two sets (i.e. $CLASS$ and $PROPERTY$) in order to create a new set of ordered pairs (i.e. $HAS\_PROPERTIES$).

- $HAS\_INSTANCES$ relationship is defined to attach each class to its sets of instances. The relation operator $\leftrightarrow$ is used create the new set of ordered pairs $HAS\_INSTANCES$.

- $HAS\_VALUES$ relationship associates a value to a pair $(instance, property)$ using the $\leftrightarrow$ relation operator. This pair is formalized using *Cartesian product* Event-B notation $\times$ which allows the definition of a set of pairs whose first part is in $INSTANCE$ and second part is in $PROPERTY$.

**Formalization of the $Is\_a$ relationship.** The $Is\_a$ relationship is a fundamental relation that links the classes together in a class hierarchy. More precisely, it defines the subsumption relationship provided by the inheritance relation classically defined in object oriented languages.

- **XML-OWL description of the $Is\_a$ relationship.**

  The OWL description of the $Is\_a$ relationship is given by the XML description below. It defines the fundamental class constructor *subClassOf* which relates a specific class to a more general one (here defines the current class as a subclass of *MotherClass*) and hence allows classes subsumption.

```
<owl:Class rdf:ID="SubClass">
    <rdfs:subClassOf rdf:resource="MotherClass" />
</owl:Class>
```

- **Event-B formalization of the $Is\_a$ relationship.**

  Listing 4.3 depicts the Event-B formalization of the $Is\_a$ relationship (subsumption) in $axm4$. First, a set $IS\_A$ gathers all the possible $IsA$ relations (subsets) and $IsA$ is modeled as a relation between classes using the $\leftrightarrow$ relation Event-B operator. A second part of this definition precises the semantic of an $IsA$ relationship by formalizing a constraint associated to inheritance i.e. inclusion of sets of instances. Indeed, $axm1$ explicitly states that the set of instances of a class $x$ such that $x$ *IsA* $y$ is included in the set of instances of a class $y$.
  The Event-B notation $x \mapsto y \in IsA$ is used to characterize a pair belonging to an $IsA$ relation ($\mapsto$ being the operator associating for combining two elements to form an ordered pair).
  The instances of $x$ and $y$ classes are cached using the Event-B notation $ran(x \lhd r)$. $\lhd$ refers to a *domain restriction* of a relation $r$ on $x$ (restrict $r$ so that it only contains pairs whose first part is $x$) and $ran$ refers to the range of the restricted set $r$ i.e. the set of second parts of all the pairs in restricted set $r$.

Listing 4.3: The Is_a relationship.

```
Context Ontology_Relations

...

Axioms

axm4: IS_A ={IsA | IsA ∈ CLASS ↔ CLASS ∧

        (∀ x, y·(x ∈ CLASS ∧ y ∈ CLASS ∧ x↦y ∈ IsA
        ⇔
        union({r· r ∈ HAS_INSTANCES| ran({x}◁r)})
        ⊆ union({r· r ∈ HAS_INSTANCES|
        ran({y}◁r)}) ))
        }
```

#### 4.1.3.2 Non-canonical concepts

Once the canonical concepts are defined, ontology modeling languages of the literature like OWL or Plib offer the possibility to define (derive) other ontological concepts from the canonical ones. These concepts are defined as non canonical concepts.

To define such non canonical concepts, ontology languages offer various derivation operators. Below, we review the main operators studied in the context of this thesis.

**Equivalence.**

- **XML-OWL description of the equivalence relationship.**

  The following XML description gives the OWL definition of the equivalence relationship between two classes. It depicts the *equivalentClass* property which is formalized to specify that two classes has the exact same set of instances.

```
<owl:Class rdf:about="Class">
   <equivalentClass rdf:resource="EquivalentClass"/>
</owl:Class>
```

- **Event-B formalization of the equivalence relationship.**

  Listing 4.4 gives the formalization of the *Equivalence* relationship. As for the $Is\_a$ relationship, the equivalence is defined as a relation between classes (i.e. $CLASS \leftrightarrow CLASS$). Its semantics is defined in $axm5$ first, at $CLASS$ level to state that the defined relation is reflexive, symmetric and transitive and second, at instance level to state that two equivalent classes have the exact same set of instances.

Listing 4.4: The *Equivalence* relationship.

```
Context Ontology_Relations

...


Axioms
  axm5: EQUIVALENCE = { EQo| EQo ∈ CLASS ↔ CLASS
        ∧
        (∀ x· (x ∈ CLASS ⇒ x↦ x ∈ EQo))
        ∧
        (∀ x, y· (x ∈ CLASS ∧ y ∈ CLASS ∧ x↦ y ∈ EQo ⇒ y↦ x ∈ EQo))
        ∧
        (∀ x, y, z· (x ∈ CLASS ∧ y ∈ CLASS ∧ z ∈ CLASS ∧ x↦ y ∈ EQo ∧ y ↦ z ∈ EQo
                              ⇒ x↦ z ∈ EQo))
        ∧
        (∀ x, y· (x ∈ CLASS ∧ y ∈ CLASS ∧ x↦ y ∈ EQo
        ⇔
                union({r· r ∈ HAS_INSTANCES| ran({x}◁r)})
                = union({r· r ∈ HAS_INSTANCES|
                ran({y}◁r)}) ))
        }
```

**Restriction.** The restriction operator is applied on a class or on a class expression. It defines a filter on the population (instances) of a class using a logical property.

- **XML-OWL description of the restriction operator.**

  The restriction operator considers a class, a property and defines a restricted population for a class. Three kinds of restrictions on a property are defined in OWL: *allValuesFrom*, *someValuesFrom* and *cardinality*. We focused on *allValuesFrom* restriction and its OWL description is given in the following XML description. It states that for each instance of the restricted class that valuates the specified property (using *onProperty* OWL construct), the value of this property has to be one of those specified in the restriction (using the *allValuesFrom* OWL construct).

```
<owl:Restriction>
<owl:onProperty rdf:resource="..." />
<owl:allValuesFrom rdf:resource="..."  />
</owl:Restriction>
```

- **Event-B formalization of the restriction operator.**

  Listing 4.5 shows the formalization of the restriction operator. It is defined as a filter on the set of instances. First, the filtering property is defined as a higher order functional parameter on the instances using the $Property\_Verification$ function on instances in $axm6$. This function is then instantiated and parameterized regarding a specific ontological property and its possible values (same as $onProperty$ construct in OWL) using a lambda function.

Listing 4.5: The *restriction* operator.

```
Context Ontology_Relations

...

Axioms
  ...
  axm6: Property_Verification = INSTANCE ⟶ BOOL
  axm7: RESTRICTION = {rest| rest ⊆ INSTANCE ∧
          (∃ p. (p ∈ Property_Verification ∧ rest= p^−1[{TRUE}]))
          }
```

$axm7$ states that the obtained restriction is defined as the set of instances whose value, by the function $Property\_Verification$, equals to $TRUE$. The inverse function is used for this purpose, it returns a set of $INSTANCE$ that are defined as first part of the $Property\_Verification$ pairs and for which the second part of the pair is $TRUE$. Thus, only the instances that satisfied the property restriction (parametrized in an instance of $Property\_Verification$) are contained in a *rest* set.

**Union.** OWL provides additional constructors with which to form classes. Thus, the binary union class is defined to collect the instances of two classes using the $unionOf$ OWL construct.

- **XML-OWL description of the union relationship.**

  The XML-OWL definition for the union relationship is defined on two classes as follows.

```
<owl:Class rdf:ID="UnionClass">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="Class1" />
    <owl:Class rdf:about="Class2" />
  </owl:unionOf>
</owl:Class>
```

- **Event-B formalization of the union relationship.**

  Like for the restriction operator, the union relationship builds explicitly the set of instances as the union of the sets of instances of the two classes.

Listing 4.6: The *union* relationship.

```
Context Ontology_Relations

...

Axioms
  axm8: UNION_OF = {unionOf|
          (unionOf ∈ CLASS × CLASS ↔ CLASS))
          ∧
          (∀ x, y, z·(x ∈ CLASS ∧ y ∈ CLASS ∧ z ∈ CLASS ∧ x ↦ y ↦ z ∈ unionOf
              ⇒
              ∀ instance· (instance ∈ INSTANCE
                ⇒
                ∃ hasInstance· (hasInstance ∈ HAS_INSTANCES ∧ (x ↦ instance ∈ hasInstance
                ∨ y ↦ instance ∈ hasInstance))
                ⇒
                z ↦ instance ∈ hasInstance))))
            ) }
```

Listing 4.6 shows the formalization of the union relationship. The $UnionOf$ relationship is defined as a relation between two classes. The defined logical property states that if an instance belongs to a class $x$ or an instance belongs to a class $y$ then it belongs to the class $z$ belonging to the $UnionOf$ relation.

#### 4.1.3.3 Ontological relationships composition

Ontological relationships composition is not handled in the ontological relationships defined above. However, a track to introduce such a composition operator in our Event-B ontology modeling language is possible by handling sets of *CLASS* rather than a single *CLASS*. As example, we show on listing 4.7 the formalization of the union relationship which handles relationships composition. Thus, this *unionOf* is defined as a binary relationship between two sets of classes using the Powerset $\mathbb{P}$ Event-B operator (bold in Listing 4.7 instead of between two classes as in listing 4.6). Therefore, the left side of *unionOf* can be obtained by applying another ontological relation on the ontological classes. For example, we can define a union between a equivalence relationship and a subsumption relationship similar to

$$\text{EQo}[\{x\}] \mapsto \text{Isa}[\{n\}] \in \text{unionOf}$$
$$\text{Assuming that EQo}[\{x\}] = \{y \ / \ x \mapsto y \in \text{EQo}\} \in \mathbb{P}\ (\text{CLASS})$$
$$\text{and Isa}[\{n\}] = \{z / \ n \mapsto z \in \text{Isa}\} \in \mathbb{P}\ (\text{CLASS}).$$

Listing 4.7: The *union* relationship.

```
Context Ontology_Relations

...

Axioms
  axm8: UNION_OF = {unionOf|
          (unionOf ∈ ( ℙ(CLASS) × ℙ(CLASS)↔ CLASS))
          ∧
          (∀ x, y, z·(x ∈ ℙ(CLASS) ∧ y ∈ ℙ(CLASS) ∧ z ∈ CLASS ∧ x↦y↦z ∈ unionOf
          ⇒
          ∀ instance· (instance ∈ INSTANCE
            ⇒
            ∃ hasInstance· (hasInstance ∈ HAS_INSTANCES ⇒
            (∀ n, m· (n ∈ x ∧ m ∈ y ∧ (n↦instance ∈ hasInstance ∨ m↦instance ∈ hasInstance))
            ⇒
            z↦instance ∈ hasInstance))))
          ) }
```

**Remark.** Above, we have shown the event-B formalization of the ontological constructs we focused on in our thesis but, in the same manner, the *Ontological_Relations* Event-B context can be extended and other ontological constructs can be formalized.

Our work focuses on engineering application domain that uses modeling languages with a classical semantics using Closed World Assumption (CWA). Thus, the corresponding ontologies are defined in a CWA. The proposed ontology language has been formalized using only the Event-B constructs and the corresponding set theory. However, a full deep modeling approach where the mathematical constructs are rewritten and Open World Assumption (OWA) introduced can be handled. This last topic is out of scope of our work.

### 4.1.4 An example of ontologies

Once the ontology concepts are defined at a generic level, it is possible to define specific ontologies as instances of the generic concepts. Hence, the Event-B context defining the specific ontology is defined as an extension of the previous context defining the ontology modeling concepts.

#### 4.1.4.1 Ontology for diplomas: *Is_a* and equivalence

Listing 4.8 gives an extract of the *Diplomas* ontology we have formalized as instances of the generic concepts previously introduced. The defined ontology illustrates the $Is\_a$ and equivalence relationships. Two parts compose this definition.

- an axiomatization part defined by the axioms clauses (*axm1, axm2, axm3*). It defines the extension of the sets corresponding to instances of the concepts defined in the generic context;

- a theorem part (*thm1, thm2*), which represents properties to be proved. The theorems guarantee that the defined relations at the instance level are conform to their definition at the generic level.

Listing 4.8: An ontology example.

```
Context Diplomas_Ontology
Extends Ontology_Relations

Constants IsA, EQ, Diploms, Bachelor, Master, Engineer, Phd

Axioms
        axm1: partition(CLASS,
                {Diploms},
                {Bachelor},
                {Master},
                {Engineer},
                {Phd}
                )
        axm2: isA = {
                Master↦ Diploms,
                Bachelor↦ Diploms,
                Engineer↦ Diploms, Phd↦ Diploms
                        }
        axm3: EQ ={
                Bachelor↦Bachelor, Master↦Master,
                Engineer↦Engineer, Phd↦Phd,
                Master↦Engineer, Engineer↦Master
                        }

// Relevant Theorems

        thm1: isA ∈ IS_A
        thm2: EQ ∈ EQUIVALENCE
```

The above ontology defines classes for diplomas. Other classes subsumed by the diploma class are defined: Bachelor, Master, Phd. The subsumption is materialized by the definition of the $isA$ set. An equivalence relation $EQ$ is also defined. It states first, that each defined class is equivalent to itself (reflexivity property) and second, that $Master$ and $Engineer$ diplomas are equivalent.

Proving the theorems guarantees that the defined instances is correct regarding their generic definitions in the *Ontology_ Relations* context.

#### 4.1.4.2 Ontology for diplomas: use of the restriction operator

As depicted in Listing 4.9, we use the Diplomas case study to illustrate how the restriction operator is applied for a specific ontology.

We want to define a filtered set of engineers students whose level of study (*level* property)equals 5. A restriction is used for this purpose.

Listing 4.9: An example of ontology.

```
Context Diplomas_Ontology
Extends Ontology_Relations

Constants
        Bachelor, Master, Engineer, PhD,
        Diplom, LMDDiplom, ClassicalDiplom,   // Classes
        SI, SRLC, ENSEEIHT, ISAE, Person
        BachelorStudent, MasterStudent,      // Instances
        level, age, name,                 // Properties

        hasInstances, hasValues, LevelEng, RestrictionEngineer5     // Instances of Generic concepts

Axioms

axm1: partition(CLASS,
                {Bachelor},
                {PhD},
                {Master},
                {Engineer},
                {Diplom},
                {LMDDiplom},
                {ClassicalDiplom}
        )
axm2: partition(INSTANCE,
                {SI},
                {SRLC},
                {ENSEEIHT},
                {ISAE},
                {BachelorStudent},
                {MasterStudent}
        )
axm3: partition(PROPERTY,
                {level},
                {age},
                {name}
        )
axm4: hasInstances = {
                Master↦SI,
                Master↦SRLC,
                Engineer↦ENSEEIHT,
                Engineer↦ISAE,
                Diplom↦ENSEEIHT,
                Person↦BachelorStudent,
                Person↦MasterStudent
        }
axm5: hasValues = {
                SI↦level↦5,
                ENSEEIHT↦level↦5
        }
axm6: LevelEng = (
                λi. i ∈ INSTANCE ∧ hasInstances^−1[{i}]={Engineer} |
                                        bool(hasValues(i↦level)=5))
axm7: RestrictionEngineer5 = LevelEng^−1[{TRUE}]

\\Relevant theorems

thm1: hasInstances ∈ HAS_INSTANCES
thm2: hasValues ∈ HAS_VALUES
thm3: LevelEng ∈ Property_Verification
thm4: RestrictionEngineer5 ∈ RESTRICTION
...
```

The previous context extends the generic context as follows.

- $CLASS$, $PROPERTY$ and $INSTANCE$ abstract carrier sets are populated in $axm1$, $axm2$ and $axm3$;

- A lambda function is introduced in $axm6$ to parametrize a property restriction and formalize restriction on the instances $i$ that valuated the *level* property. It states that *level* value has o be equal to 5;

- $RestrictionEngineer5$ set defined in $axm7$ gathers the instances that satisfy the property restriction defined in $axm6$;

- $thm3$ asserts that this instantiation produces an instance of the $Property\_verification$ parameter of the restriction operator defined at the generic level ($thm3$);

- $thm4$ states that we have defined a restriction corresponding to the definition of the abstract level.

Finally, discharging $thm3$ and $thm4$ will guarantee the correctness of this restriction regarding its generic definition in the *Ontology_Relations* context.

### 4.1.5 Deduction rules and theorems

In practice, when ontologies are defined, users set up reasoners in order to infer knowledge from the canonical and non-canonical reasoning. These reasoners apply deduction rules in order to infer new facts or to define classification trees (placement). The application of these rules is performed within reasoning engines.

However, the correctness of the applied deduction rules as well as the inferred facts (regarding the applied deduction rules) can be questioned. The validation of the correctness of these rules is mandatory in order to guarantee the correctness of the performed reasoning and thus of the reasoners.

The definition of these deduction rules corresponds to theorems in Event-B. The work we have achieved by formalizing deduction rules as theorems led us to the capability to prove, in the defined Event-B generic context, theorems corresponding to deduction rules used by reasoners.

As example, we consider the theorem used by reasoners to build class hierarchy in order to make explicit the subsumption relationship (placement in the subsumption hierarchy) for non-canonical classes. The following theorem *thmx* deals with the restriction operation on classes. It states that if two restriction classes *C1* and *C2* are defined on the same class $C$ and $P1$ property implies $P2$ property ($P1 \Rightarrow P2$) then a new $Is\_a$ relation can be inferred between $C1$ and $C2$.

$$Thmx : (C1 : Restriction(C, P1) \wedge C2 : Restriction(C, P2) \wedge P1 \Rightarrow P2) \Rightarrow C2\ Is\_a\ C1$$

Listing 4.10: Deduction theorems.

*thm*1: ∀ x, p1, p2· (x ∈ INSTANCE ∧ p1 ∈ Property_Verification ∧ p2 ∈ Property_Verification
∧ (p1(x)=TRUE ⇒ p2(x)=TRUE)
⇒
$p2^{-1}$[{TRUE}] ⊆ $p1^{-1}$[{TRUE}]))

*thm*2: ∀ x, p1, p2· (x ∈ INSTANCE ∧ p1 ∈ Property ∧ p2 ∈ Property ∧ (p1(x) ⇒ p2(x))
⇒
(∀ y, z, hasInstances, isA· (y ∈ CLASS ∧ z ∈ CLASS ∧ hasInstances ∈ HAS_INSTANCES
∧ isA ∈ IS_A ∧ hasInstances[{y}] = $p1^{-1}$[{TRUE}] ∧ hasInstances[{z}] = $p2^{-1}$[{TRUE}]
⇒
y↦z ∈ isA )))

Listing 4.10 depicts the formalization of $Thmx$ in Event-B. Theorem $Thmx$ is split into two sub-theorems. A first lemma, $thm1$ states that if two sets of instances satisfy properties $P1$ and $P2$ respectively such that $P1 \Rightarrow P2$ then the former set of instances is included in the later. Theorem $thm2$ defines the suited theorem. Lemma $thm1$ is used to prove this theorem.

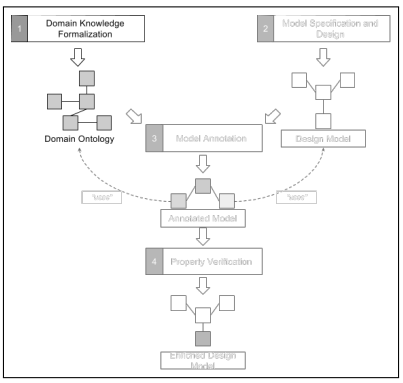
Finally, note that the proof of this theorem can be used to justify or assert that the rule implemented by the reasoners is valid.

## 4.2 Strengthening design models using domain models: an annotation based approach

In this section, we show how the proposed stepwise methodology for design models strengthening can be deployed in the case of formal modeling. The refinement and proof Event-B formal method has been chosen for this purpose. It applies the four defined steps (chapter 2 - section 2.3) and gives the root Event-B models for the diplomas case study.

### 4.2.1 Step 1. Domain knowledge formalization

As discussed above, all the basic ontological concepts and relationships are formally described within a generic Event-B context. This context can be extended to be specialized for a specific ontology.



### An ontology for the *Diplomas* case study

The *Diplomas* ontology has been already given in previous section, an extract of it is depicted in Listing 4.11 to show the end-to-end application of our global design models strengthening approach. The *Ontology_Relations* context (defined in section 4.1) is extended by *Diplomas_Ontology*. Diplomas are defined as classes. The equivalences between the different classes are explicitly formalized using the specific equivalence relation EQo belonging to the set EQUIVALENCE of equivalence relationships. The correct definition of EQo relationship is guaranteed by proving the theorem in *thm*3. This proof requirement entailed by the use of formal methods guarantees that used specification relationships like EQo formally fulfill the equivalence relationship properties.

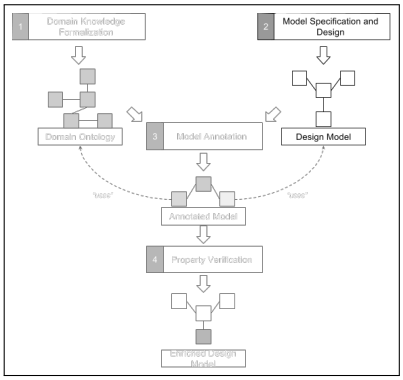
Listing 4.11: Diplomas ontology.

```
Context Diplomas_Ontology
Extends Ontology_Relations

Constants Master, Engineer, Bachelor, PhD

Axioms
  axm1: partition(CLASS, {Master}, {PhD}, {Bachelor}, {Engineer})
  axm2: EQo = {Bachelor↦Bachelor, Engineer↦Engineer, PhD↦PhD,
  Master↦Master, Master↦Engineer, Engineer↦Master}
  thm3: EQo ∈ EQUIVALENCE      Theorem
End
```

In addition, the *Diplomas_Ontology* context describes *Master, Bachelor, PhD, Engineer* as specific diplomas classes. It also states that an *Engineer* diploma is equivalent to a *Master* diploma.

## 4.2.2 Step 2. Model specification and design

Design models are formalized as classical Event-B models with contexts and machines. Static part (constants, types and data) of the design models is defined within contexts and the dynamic part is referred to as a machine which *sees* the defined contexts (static part). Listing 4.12 depicts the generic design model context as follows.




- *CONCEPT*, *ATTRIBUTE*, *MODEL_INSTANCE* and *TYPE* relevant sets characterize the concepts, attributes, instances and data types involved in the definition of a design model.

Listing 4.12: Generic design model context.

```
Context Generic_Design_Model

Sets CONCEPT, ATTRIBUTE, MODEL_INSTANCE, TYPE

Constants HAS_ATTRIBUTES, NATURAL, BOOLEAN, INTEGER, STRING,
HAS_VALUE_FOR_TYPE_BOOLEAN,HAS_VALUE_FOR_TYPE_INTEGER,
HAS_VALUE_FOR_TYPE_NATURAL, HAS_VALUE_FOR_TYPE_STRING, one, two, three, four, five, ...

Axioms
  axm1: HAS_ATTRIBUTES = CONCEPT ↔ ATTRIBUTE
  axm2: TYPE = NATURAL ∪ BOOLEAN ∪ INTEGER ∪ STRING
  axm3: NATURAL = {one, two, three, four, five}
  axm4: HAS_VALUE_FOR_TYPE_INTEGER = HAS_ATTRIBUTES → INTEGER
  axm5: HAS_VALUE_FOR_TYPE_BOOLEAN = HAS_ATTRIBUTES → BOOLEAN
  axm6: HAS_VALUE_FOR_TYPE_NATURAL = HAS_ATTRIBUTES → NATURAL
  axm7: HAS_VALUE_FOR_TYPE_STRING = HAS_ATTRIBUTES → STRING
  ...
End
```

- *HAS_ATTRIBUTES* relationship ($axm1$) is defined to link each concept with its attributes.

- Four possible data types are defined: NATURAL, BOOLEAN, INTEGER or STRING their formalization is given in $axm2$ and $axm3$.

- *HAS_VALUE_FOR_TYPE_INTEGER*, *HAS_VALUE_FOR_TYPE_BOOLEAN*, *HAS_VALUE_FOR_TYPE_NATURAL* and *HAS_VALUE_FOR_TYPE_STRING* are defined in $axm4$, $axm5$, $axm6$ and $axm7$ to associate each concept's attribute with its data type.

Note that the above defined context is generic and needs to be extended to define specific applicative contexts. The explicit definition of these generic entities (at a meta level) allows the definition of generic annotation mechanisms. These mechanisms are then used to strengthen design models as shown in the next section.

### A design model for the *Diplomas* case study

The static part associated to the students information system case study is given in the *student_Model* context (Listing 4.13) as follows.

Listing 4.13: Student design model context.

```
Context Student_Model Extends Generic_Design_Model

Constants m, p, e, b, titi, toto, DIPLOMS, STUDENTS, PREVIOUS_DIPLOMA
//Master, PhD, Engineer and Bachelor diplomas

Axioms
  axm1 : CONCEPT = DIPLOMS ∪ STUDENTS
  axm2 : DIPLOMS ∩ STUDENTS = ∅
  axm3 : partition(DIPLOMS, {e},{m},{p},{b})
  axm4 : partition(STUDENTS, {toto},{titi})
  axm5 : PREVIOUS_DIPLOMA = STUDENTS ↔ DIPLOMS
  axm6 : finite(CONCEPT)
End
```

- *DIPLOMS* and *STUDENTS* concepts are introduced as subsets of the generic carrier set *CONCEPT* in *axm*1 and *axm*2;
- *e*, *b*, *m*, *p* constants are defined as *DIPLOMS* (elements of *DIPLOMS* set) in *axm*3. These constants respectively refer to Engineer, Bachelor, Master, PhD diplomas;
- *toto* and *titi* are defined as *STUDENTS* in *axm*4;
- Finally, *axm*6 asserts that the *CONCEPT* carrier set is a finite one.

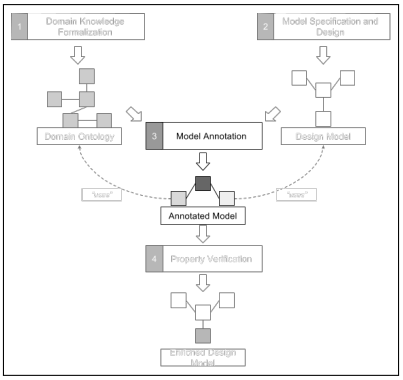
Once the different concepts are described, it becomes possible to describe the behavioral part of the model. Indeed, the model considers the case of a student willing to register for a PhD. For the case study, the dynamic part (Listing 4.14) defines the *Register* event within an Event-B machine to allow a student to register for a PhD. An invariant *inv1* ensures that a student can register for a *PhD* only if he/she holds a master degree.

Listing 4.14: Student design model machine.

```
Machine Student_Register

  Invariants
    inv1 : ∀ x· (x ∈ STUDENTS ∧ x↦ p ∈ phd_register ⇒ (previousDiplom[{x}] ⊆ {m}) ∧
          previousDiplom[{x}] ∩ {m} ≠ ∅)

  Events

    Phd_Register ≙ Any Dip

      Where
        grd1: dip ∈ {m}
        grd2: previousDiplom[{student}]={dip}

      Then
        act1: phd_register = phd_resgiter ∪ {student ↦ p}

End
```

### 4.2.3 Step 3. Model annotation

Listing 4.15 depicts the formal definition of generic annotation relationships within an Event-B context as follows.




- An *ANNOTATION_CLASS* linking concepts of design models to classes of ontologies is formalized in $axm1$;
- *ANNOTATION_PROPERTIES* relation is defined in $axm2$ to link properties of an ontology to attributes of a design model;
- In the same way, $axm3$ formalizes *ANNOTATION_INSTANCES* relation in order to annotate design model instances with ontology instances.
- Finally, the *Annotation_Relationship* can be extended to define other kinds of annotation relationships, for example to link constraints, data types etc.

Listing 4.15: Annotation relationship formalization context.

```
Context Annotation_Relationship
Extends Ontology_Relations, Generic_Design_Model

Axioms
  axm1: ANNOTATION_CLASS =
              CLASS ↔ CONCEPT
  axm2: ANNOTATION_PROPERTIES = PROPERTY ↔ ATTRIBUTE
  axm3: ANNOTATION_INSTANCES = INSTANCE ↔ MODEL_INSTANCE
  ...
End
```

The above *Annotation_Relationship* context is extended to define specific design model annotations. Once these annotations are defined, they can be integrated into the design model and enrich it. Thus, annotations are exploited to access the ontological concepts and properties in design models. Moreover, note that using the defined annotation relationships will link only the specified model concepts to the ontology (when instantiating the annotation relationship). In this way, models are separated from the domain model and thus both ontologies and design models can evolve separately and asynchronously.

**Remark.** Note that the above presented generic annotation relationship could be parameterized and constrained using axioms to define a more specific kind of annotations (like Case_of, inheritance, etc.).

### The *Diploma* case study annotation

Listing 4.16 defines annotations for the $m$ and $e$ concepts manipulated at the design model level. $axm1$ states that $m$ and $e$ are annotated by the $Master$ and $Engineer$ ontological classes respectively. Theorem $thm1$ is added to ensure that the formalized *annotation* relation is indeed an instance of *ANNOTATION_CLASS*.

Listing 4.16: Annotation model context.

```
Context Diplomas_Annotations
Extends Annotation_Relationship

Axioms
  axm1: annotation = {Master↦m, Engineer↦e}
  thm1: annotation ∈ ANNOTATION_CLASS      Theorem

End
```

When the annotation relation is established on the design model (Listing 4.14) the annotated student design model is obtained. It becomes possible to borrow, in the design model, the reasoning obtained from the equivalence of $Master$ and $Engineer$ concepts (or any other ontological relationship that may exist between $Master$ and $Engineer$ classes).

Listing 4.17 depicts the *Student_Register* machine after the annotation process. The invariant *inv1* is rewritten (in bold) to integrate the annotations. Indeed, it states that any student holding a previous diploma that is equivalent to the inverse annotation of $m$ can be registered for preparing a $PhD$ (here $Engineer$ but the ontology can evolve and other kind of diplomas equivalent to $Master$ can be formalized in the ontology and thus available at the design model level - even after annotation). Using the annotation relationship, it becomes possible to borrow knowledge from the domain ontology and explicit the implicit relationship that may exists between design model concepts (here the equivalence between $m$ and $e$).

Thus, the equivalence ontological relationship can now be exploited to enrich the design model.

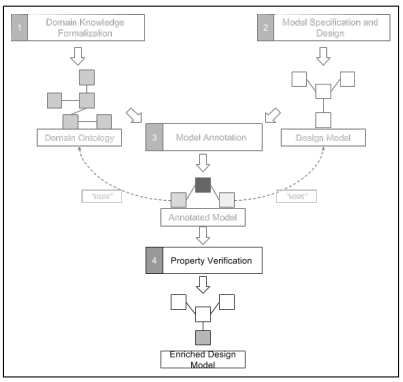
Listing 4.17: Annotated Student_Register machine.

```
Machine Student_Register

Invariants   inv1 : ∀ x· (x ∈ STUDENTS ∧ x↦ p ∈ phd_register ⇒ ((previousDiplom[{x}] ⊆ {m})
                  ∨ (previousDiplom[x] ⊆ annotation[EQo[annotation⁻¹[m]]]) ))
```

```
Events

        Phd_Register ≙ Any Dip

        Where
              grd1: dip ∈ {m,e}
              grd2: previousDiplom[{student}]={dip}

        Then
               act1: phd_register = phd_resgiter ∪ {student ↦ p}

End
```

### 4.2.4 Step 4. Properties verification

The property verification step is achieved trough discharging all the proof obligations (PO). In fact, once the annotation step is achieved new information (and reasoning) becomes available in the design model. Thus, new PO may be generated and the old ones my become no longer valid regarding the new imported domain information. A verification step where all the required proofs are (re)-checked is mandatory to ensure the correctness of the design model.




#### The *Diploma* case study verification

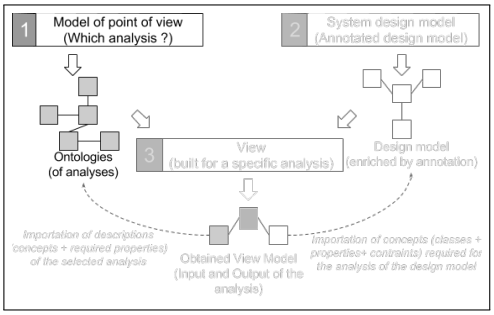
The new definition of invariant *inv*1 of Listing 4.17 uses *explicitly* the defined annotation relationship. Invariants similar to *inv*1 are defined using the annotation relationship. For our case study, the correctness of the new enriched model is proven. The new POs generated by the annotation are discharged by proving that invariant *inv1* still holds after the *Phd_Register* event is triggered. All the POs associated to the *Student_Register* machine have been proved for all values of *dip* that are equivalent to *Master*.

## 4.3 Multi-view modeling

We show below how the proposed stepwise multi-view modeling methodology can be deployed in the context of Event-B formal method setting. The Diplomas case study is used at each step of the deployment for illustrative purpose.

### 4.3.1 Step 1. Model of point of view

The different concepts describing the main features of a point of view model are defined. All the required properties and relationships for building a point of view are formally described within a generic Event-B context. This context can be extended to be specialized for a specific point of view. Listing 4.18 gives an extract of the Event-B *Generic_POV* context defined as follows.




Listing 4.18: Generic point of view context.

```
Context Generic_POV
Extends Ontology_Relations
Sets POV_CLASS, VIEW_PROPERTY, USED_METHOD, CONSTRAINTS, CONFIG_INFO
Constants REQUIRED_PROPERTY, POV_CONSTRAINTS, POV_PROPERTIES, POV_CONFIGURATION
Axioms
  axm1: REQUIRED_PROPERTY ⊆ PROPERTY
  axm2: POV_PROPERTIES = POV_CLASS ↔ REQUIRED_PROPERTY
  axm3: POV_CONSTRAINTS = POV_CLASS ↔ ℙ(CONSTRAINTS)
  axm4: POV_CONFIGURATION = POV_CLASS ↔ CONFIG_INFO
  ...
End
```

- *POV_CLASS*, *VIEW_PROPERTY*, *USED_METHOD*, *CONSTRAINTS* and *CONFIG_INFO* relevant finite sets are defined. They respectively refer to the *point of view* class, view properties, used method constraints and additional configuration information (as described in section 3.3.1.1).

- *REQUIRED_PROPERTY* constant is defined as a subset of the ontological set PROPERTY to refer the set of properties of the system model required in order to trigger the described analysis (note that all the *REQUIRED_PROPERTY* set elements reference explicitly system model properties through annotation relationships).

- The *POV_CONSTRAINTS*, *POV_PROPERTIES* and *POV_CONFIGURATION* relationships link explicitly the *point of view* class *POV_CLASS* to its constraints, properties and configuration respectively.

### A point of view for the *Diplomas* case study

The defined point of view for the *Diplomas factory* analysis for printing students diplomas from the Student information system is depicted in Listing 4.19 as follows.

Listing 4.19: Diplomas factory point of view.

```
Context Diplomas_Factory_POV
Extends Generic_POV, Diplomas_Annotations
```

```
Constants reqProperties, pov_properties, pov_constraints, printingDiploma, PRINT_DIPLOMA,
          const1, const2, const3

Axioms
  axm1: partition(POV_CLASS, {PRINT_DIPLOMA})
  axm2: printingDiploma = annotation[{STUDENT}] → annotation[{DIPLOMA}]
  axm3: partition(CONSTRAINTS, {const1}, {const2}, {const3})
  axm4: REQUIRED_PROPERTY = {nb_credits}
  axm5: pov_properties = {PRINT_DIPLOMA ↦ reqProperties}
  axm6: pov_constraints = {PRINT_DIPLOMA ↦ {const1, const2, const3}}
  axm7: ∀ c,k· (c ∈ pov_constraints[{PRINT_DIPLOMA}] ∧
  k ∈ ATTRIBUTE ∧ k ∈ annotation_properties[{nb_credits}] ∧
        (∃a· (a ∈ CONCEPT ∧ a ∈ annotation[{STUDENT}] ∧ a ↦ k ∈ has_attributes
              ⇒
        projNat(hasValueForNat({a ↦ k})) ≥ 12)))
  thm1: pov_properties ∈ POV_PROPERTIES
  thm2: pov_constraints ∈ POV_CONSTRAINTS

End
```

- *Diplomas factory* point of view context extends the *Generic_POV* context to define a specific diplomas printing point of view. It also extends the *Diplomas_Annotations* context that formalizes the annotation relationships between the design model and the *Diplomas* ontology. Indeed, the point of view relationships are defined on the Ontology entities and their annotations rather than directly on the design model elements. Thus, both the point of view and the design model remain independent and evolve separately in an asynchronous manner.

- *PRINT_DIPLOMA* is defined as specific point of view class for the *Diplomas factory* point of view in $axm1$.

- *printingDiploma* is defined as an Event-B function ($axm2$) with students as input and it returns their printed diploma as output. It characterizes the external function to trigger in order to print students diplomas.

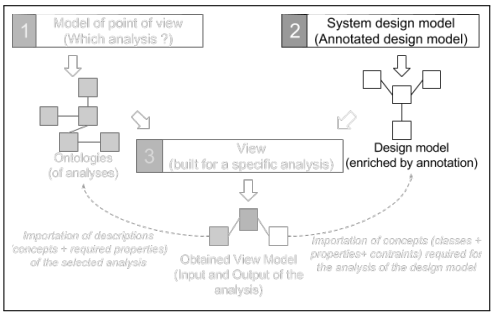
- Three POV constraints are defined in $axm3$. They are labeled as $const1$, $const2$ and $const3$. In the same way, a required property *id* defined in $axm4$.

- A POV constraint is formalized in $axm7$. It states that a student diploma can be printed using this point of view only if the student has at least 12 credits (obtained by validating enough courses).

- Theorems in $thm1$ and $thm2$ guarantee the well definition of the specified relationships regarding to the ones defined in the generic context.

### 4.3.2 Step 2. System design model

The relevant entities for the definition of a design model are formalized in a generic context as presented in Listing 4.12. This context is then extended to define specific applicative contexts.

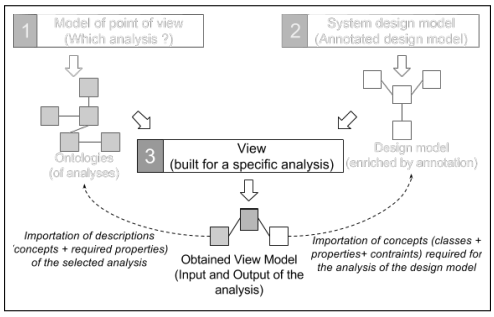



**A design model for the *Diplomas* case study**

The design model supporting multi-analyses is defined at this level. It corresponds, in this case study, to the new strengthened student information system model obtained at the end of the model annotation step (subsection 4.2.4).

### 4.3.3 Step3. View

The specific view is built by integrating the resources issued from both concepts of the point of view (step 1) and the design model (step 2). The obtained view is formalized within an event-B machine. Listing 4.20 depicts such a generic system view as follows.




Listing 4.20: Generic abstract view machine.

```
Machine Generic_View

Sees a specific point of view

Variables ...

Invariants
  inv1: ...

Events

  INITIALISATION ≙
  Then
     act1: Initialisation of model static variables
     act2: Initialisation of view variables
     ...
  End

  SpecificViewEvent ≙
  Then
     act1: trigger the view method
     act2 get the method result
  End
  ...

End
```

- *Generic_ View* machine *sees* a specific point of view to trigger the corresponding analysis.

- View properties and required properties (together with the model variables) have to be first initialized in the *INITIALISATION* Event-B event.

- The specific behavioral view (ie. triggering the external method and gathering the obtained results) is formalized within the *SpecificViewEvent* event (*act1* and *act2*).

### Diplomas factory view

A specific Diplomas factory view can be built by integrating the resources issued from both the student information system model and the *Diplomas factory* point of view. *PrintingView* abstract machine is depicted in listing 4.21. Since it sees the *Diplomas_ Factory_ POV*, both the required properties and the view properties are imported in the Diplomas factory view machine and can be exploited. A boolean variable *printed* is defined. It states whether a diploma is printed or not. It is initialized to FALSE and changed to TRUE once the diploma is printed by triggering the *PrintingDiploma* Event.

Listing 4.21: The *Diplomas factory* abstract view.

```
Machine PrintingView

Sees Diplomas_Factory_POV

Variables printed

Invariants
  inv1: printed ∈ BOOL

Events

  INITIALISATION ≙
  Then
    act1: printed := FALSE
  End

  PrintingDiploma ≙
  Then
      act1: printed := TRUE
  End

End
```

*PrintingView* machine is refined to specify the actions corresponding to diplomas printing using the *PrintingDiploma* Event. The obtained result is depicted in Listing 4.22. The *act2* adds a student-diploma pair to the printing queue. The guarantee

of the correctness of $act2$ (choosing the right student, right diploma and the right association student-diploma) is ensured by $grd1$ and $grd2$ using annotation.

Listing 4.22: The *Diplomas factory* refined view.

```
Machine PrintingView_refinement
Refines PrintingView

Sees Diplomas_Factory_POV

Variables printed, printingDip

Invariants
  inv1: printingDip ∈ printingDiploma

Events

  INITAILISATION ≙
  Then
    act1: printed := FALSE
    act2: printingDip := ∅
  End

  PrintingDiploma ≙ Refines PrintingDiploma
  Any x, y
  Where
        grd1: x ∈ annotation[{STUDENT}]
        grd2: y ∈ annotation[{DIPLOMA}]
  Then
      act1: printed := TRUE
      act2: printingDip := printingDip ∪ {x ↦ y}
  End

End
```

## 4.4 An overview of the global Event-B deployment

Figure 4.1 gives a global overview of the general Event-B deployment process of our general framework.

As presented in the previous sections, generic Event-B contexts for ontologies relationships (*Ontology_ Relations Context*), generic design models (*Generic_ Design_ Model Context*) and annotation relationships (*Annotation_ Relationships Context*) are defined. These generic contexts are extended to define specific domain ontologies (*Domain ontology Context*), design models (*Specific Design Model Context*) and design model annotations (*Design Model Annotations Context*) which explicitly link design models entities to the domain ontologies concepts. The design model machine (*Specific Design Model machine*) *sees* the *Design Model Annotations* context to exploit and integrate the annotations.

A generic point of view (*Generic_ POV Context*) is also defined. It formalizes all the properties and relationships that characterize a point of view. Both the *Design Model Annotations* context and *Generic_ POV Context* are extended to define

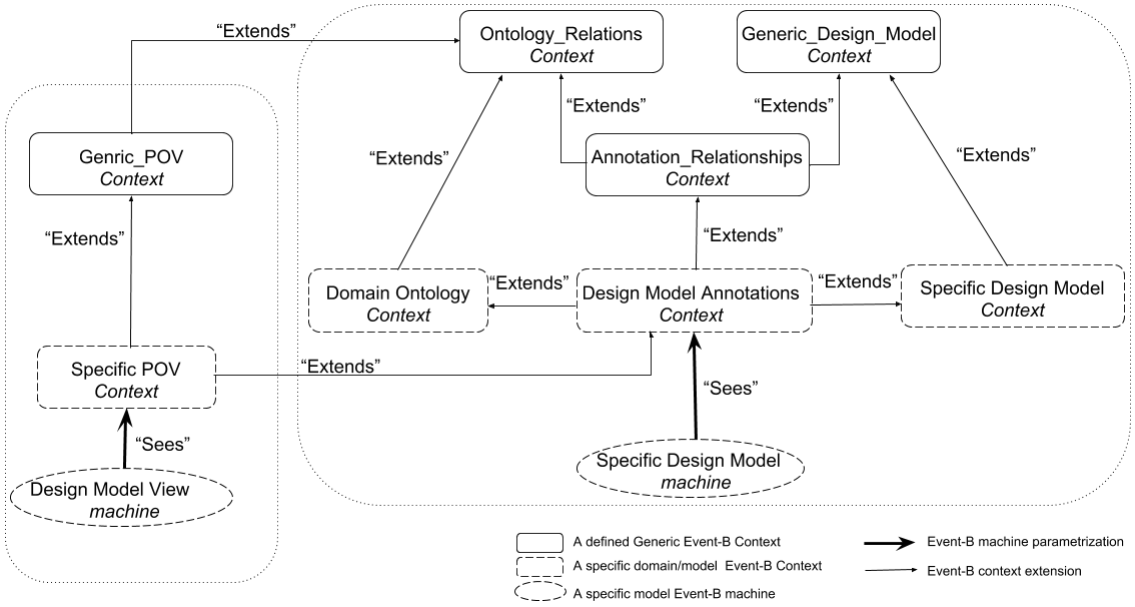



Figure 4.1: Global Event-B deployment process.

a specific point of view (*Specific POV Context*). The required properties of a point of view are defined as a subset of ontological properties (i.e. selected from a domain ontology and associated to a design model properties through annotations). The used method is defined as a function that borrows information both from the point of view and from the design model (through annotations processing). A specific view (*Design Model View machine*) sees the obtained point of view.

## 4.5 Conclusion

In this chapter, we gave the details of the Event-B formal method setting of our approach. The involved models (ontology, design model and point of view ) are formalized using set theory and predicate logic.

First we present the event B formalization of domain ontologies. Two modeling approaches have been identified and illustrated through examples. The first one is based on a shallow modeling where the concepts of ontologies are directly encoded in the target theory. In this case, the information related to the modeling language is not explicitly defined in the target formal modeling language. The second approach consists in formalizing ontology models within an Event-B context and then ontologies are defined as specific instances of the generic ontology model. The interest of the approach is that the ontology model is explicitly described and reasoning on ontologies is performed using the proof system described in the ontology model and the

underlying proof system of Event-B. In this work, we also showed an application to the validation of ontology reasoners through the definition and proof of theorems that correspond to reasoning rules in such ontology reasoners.

Our deployment of model strengthening and analysis approaches in formal method is based on deep modeling. A generic annotation relation is set to explicitly link design model entities to ontological ones. Then the domain information can be integrated incrementally and directly in the behavioral parts of the system (Event-B machine) using refinements. As result of this annotation step, the design models can be questioned, verified or checked with regards to new properties (expressed by richer invariants) exploiting annotations that borrow ontology concepts and properties to the design models.

The deployment of the multi-view analysis approach is set using Event-B contexts and machines as well. A generic point of view context is defined at a meta-level. It sets the required description to build a point of view. It can be extended to define specific point of view. Specific views are defined in Event-B machines that *sees* specific points of view. The obtained views can be refined to set more than one specific analysis in the same machine, view composition linked different interconnected views can then be defined. This case is not addressed in our thesis.

Finally, the deployed approach exploits multi-level modeling and refinement offered by the Event-B method. Thus, the asynchronous evolution of both the ontology, the design models and points of view is preserved.

# Chapter 5

# Tools implementation

This chapter describes the prototypes we have developed and the main resources used for their implementation in both MDE and formal Event-B settings.

First, we propose a Model Driven Engineering tool-chain supporting ontologies formalization, design models strengthening and multi-view analyses of design models. The MDE implementation requires the definition of the different model concepts involved in our approach at a meta-modeling level. Thus, the ontology meta-model defined in section 3.1, the annotation meta-model presented in section 3.2.3 along with the annotation mechanisms (except for inheritance that is usually offered in modeling language) and multi-view analysis meta-model of section 3.3.1 are implemented to deploy the defined approach.

Next sections present the architecture and the functionalities of the tool-chain we developed in the case of MDE setting. We also recall the different models we developed for both annotating design models by references to domain ontologies and for handling multi-view model analyses.

## 5.1 MDE context

Our approach makes an extensive use of model driven engineering techniques. To address this need, we have developed a prototype as a set of plug-ins for Eclipse based on EMF[3] and Sirius[4]. The developed tool provides graphical syntax and transformation techniques to support and ease model manipulation.

### 5.1.1 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) offers strong and large capabilities for modeling and model manipulation. Indeed, the EMF modeling platform provides editors and

code generation infrastructures. Models are built in a modular manner with referencing capabilities between models. This modularity allows a developer to use EMF with other Eclipse projects.

### 5.1.2 Sirius

Sirius is an Eclipse open source framework built on top of EMF and GMF. It provides techniques for constructing and maintaining graphical designers by specifying various model representations such as diagrams, tables and tools to edit the model. Sirius has been used in this work in order to support the whole graphical tool development we have set up as prototype.

### 5.1.3 MDE based tool creation workflow

Figure 5.1 illustrates a workflow for the construction of an MDE based tool using the Eclipse Modeling Framework, from meta-models specification to the platform integration. The workflow consists of the following steps.

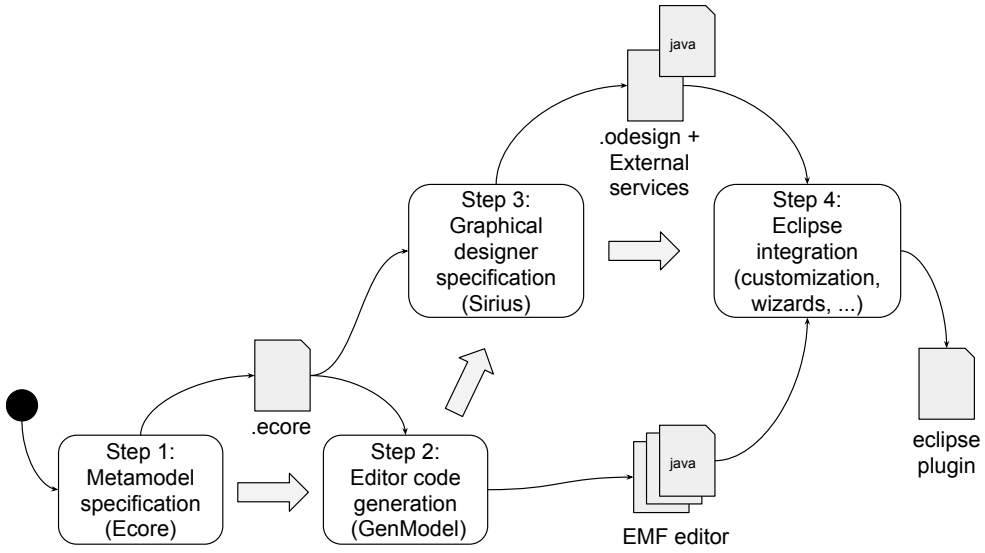



Figure 5.1: Workflow of the creation of a tool based on EMF and Sirius.

- **Step 1.** The domain meta-model is specified using EMF Ecore tool. It defines the concepts that will be created and specified in the generated editor.
- **Step 2.** Using Ecore generator (GenModel), the Java code of the EMF editor is generated.

- **Step 3.** The graphical representation of the meta-model concepts is specified using Sirius (oDesign file). In addition, tools for model edition, manipulation and transformation are implemented thanks to the external Java services.
- **Step 4.** The created tool can be customized to fit specific needs. Wizards for the creation of the project are specified to ease the integration in Eclipse platform. The created tool is distribute as an Eclipse plugin.

## 5.2 MDE based implementation

### 5.2.1 Overview of the global architecture

The prototype is implemented as a set of plug-ins for Eclipse; its architecture is illustrated in Figure 5.2. It consists of several modules grouped into four categories based on their functionality: meta-models with their EMF editors, graphical designers, model-to-model transformation (external Java services), and integration plug-ins (project wizards).

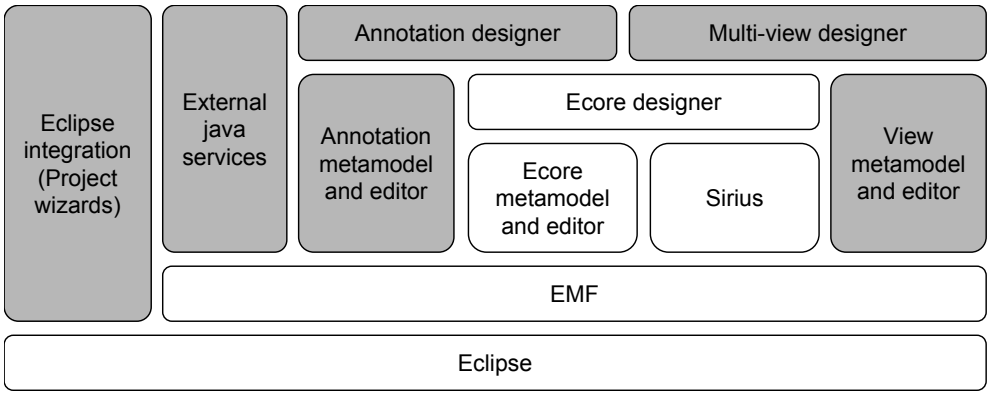



Figure 5.2: Implementation of the solution based on EMF and Sirius.

### 5.2.2 Implementation detail

Figure 5.3 describes the overall architecture of our implementation. It is composed of two main parts: first the implementation of the model annotation tool and second the implementation of the multi-view modeling tool.

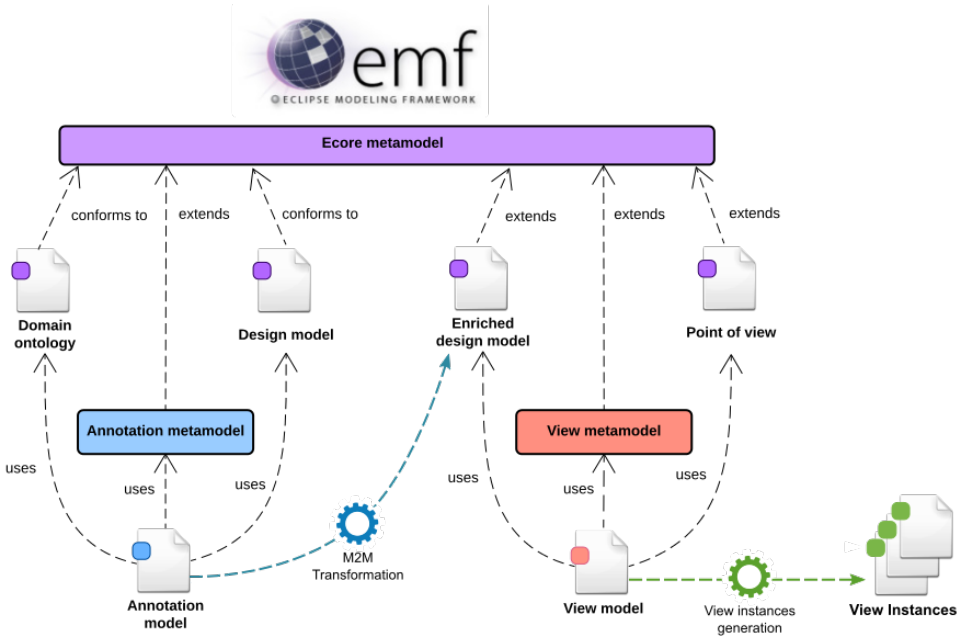



Figure 5.3: Implementation of the solution with Eclipse EMF.

### 5.2.3 The Ecore meta-model

Figure 5.4 shows the main concepts of the Ecore meta-model. The concept of classifier appears as a meta-concept allowing the definition of manipulated model concepts. From this concept, the concept of class is defined.

All the defined extensions (annotation and view meta-models) use these concepts. Relationships using references or inheritance will be set up in order to introduce the concepts relevant for our approach.

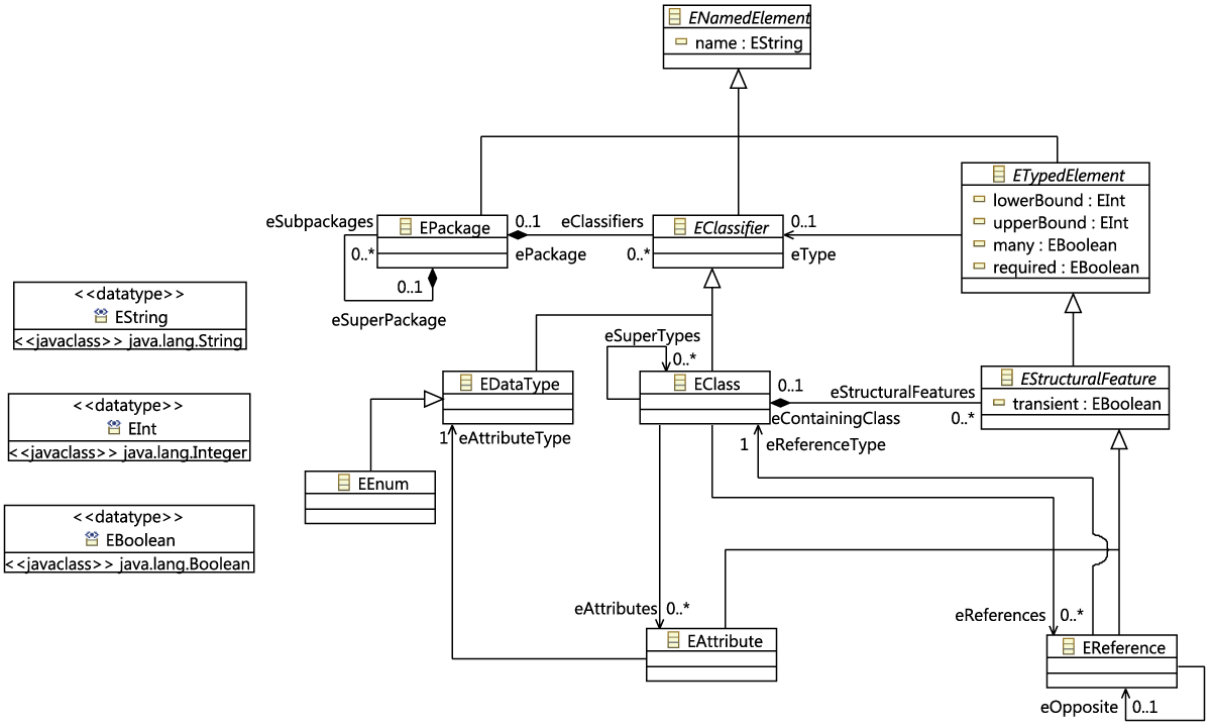



Figure 5.4: Ecore meta-model

The Ecore Meta-model is composed of the different meta-concepts allowing a developer to define classical Ecore models. Since the models we used in this thesis work and brought by the associated projects are described as UML[77] class diagrams only, we rely on the part of the Ecore Meta-model that supports the definition of the concepts involved in UML class diagrams. Classes, properties, types, references etc. are defined in the model depicted in Figure 5.4.

Note that this is not a restriction of our approach, all the concepts manipulated in the studied models can be processed in the same manner as for the concepts occurring in a class diagram.

The concepts of $EClassifier$ and $EClass$ are at the root of all the extensions we introduce in the next sections.

Next sections show how we proceeded for implementing our framework in a MDE setting based on the EMF tool chain.

#### 5.2.3.1 Implementation of the model annotation tool

The implementation of the model annotation tool starts with the definition of an annotation meta-model and the generation of the annotation editors. As depicted in Figure 5.2 the Ontology model (Domain ontology) and design model (Domain model)

are described within Ecore, therefore they conform to the Ecore meta-model. The annotation meta-model is defined as an extension of the Ecore meta-model (extends Ecore meta-model) in order to enable the annotation by *Is_ case_ of* and by Association. The integration of domain knowledge brought by the annotations into the design models is achieved through model to model (M2M) Java transformation . As result, we obtain an enriched design model as an output of this process.

### Annotation meta-model

This section recalls the first extension of the Ecore meta-model we have defined to handle model annotation. An annotation meta-model is necessary in order to introduce and to identify, through typing, the concepts of annotation.

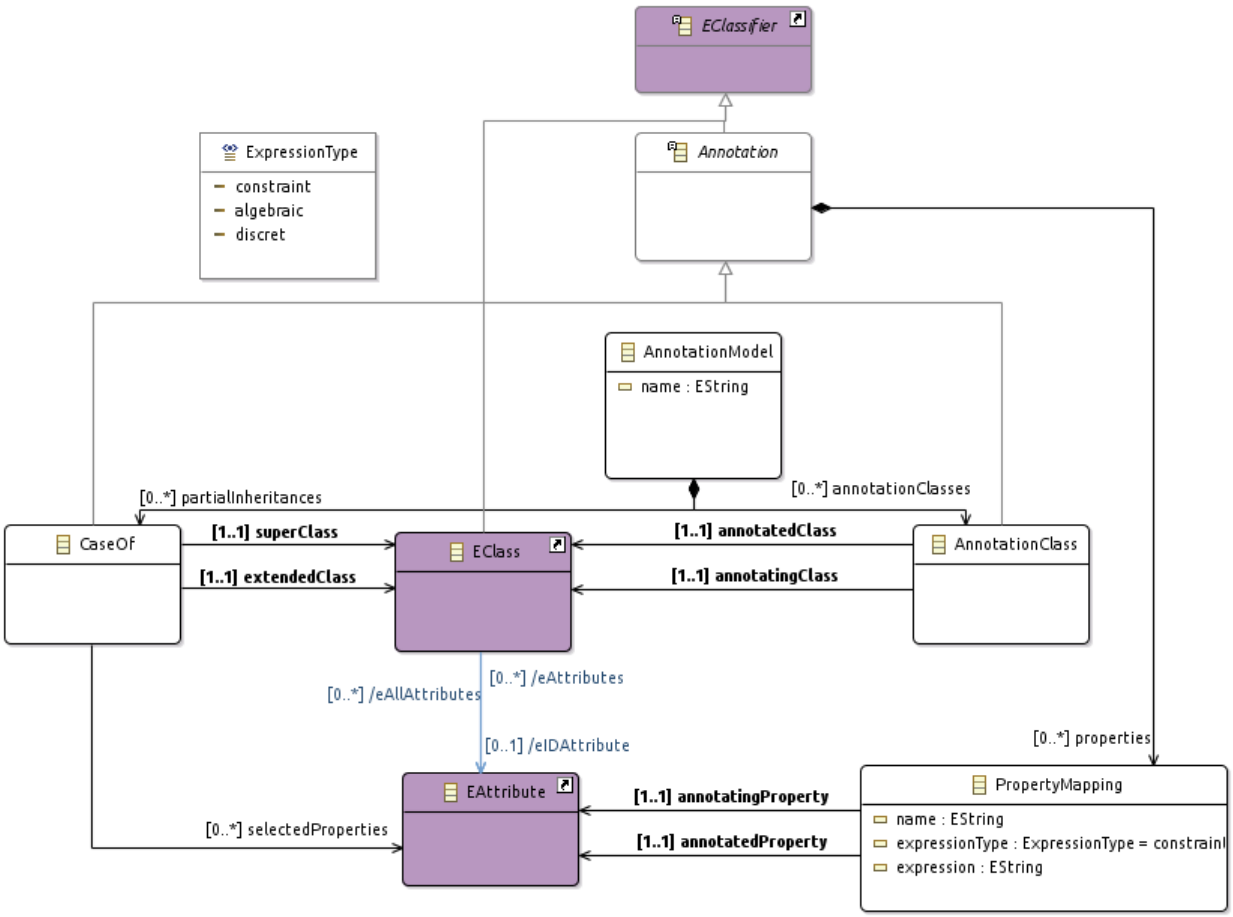



Figure 5.5: Annotation meta-model

As mentioned above, the concept of annotation is required at the meta-level in order to be manipulated through explicit typing and associated operators. The concepts

related to annotation are depicted on Figure 5.5. The following concepts have been introduced

- *AnnotationModel* is the entry point of the annotation model. It allows to traverse the annotation model. This class refers to two different resources, one for defining specific *AnnotationClass* and the other for defining the particular *CaseOf* annotation relationship.

- *Annotation* is inherited from *EClassifier* and allows to define specific annotations. It is an abstract class from which two different kinds of annotations are derived: *CaseOf* and *AnnotationClass* which respectively implement the annotation by *partial inheritance* and by *association* (see chapter 3 - section 3.2.3.2).

- *CaseOf* implements the annotation by *Is_ case_ of* mechanism. Model classes are explicitly annotated by ontology classes using *partial inheritance* relationship. Thus, the selection of ontological properties to be inherited in the annotated design model can be performed using *selectedProperties* reference.

- *AnnotationClass* implements the annotation by *association* mechanism. A model class is explicitly associated to an ontology class using this relationship. This type of annotation is directly parameterized by the user. Properties mapping to link model properties with ontology properties can be defined at this level using the *properties* composition relation (between *Annotation* and *PropertyMapping*).

- *PropertyMapping* is introduced to define properties correspondences (mapping). It handles the definition of correspondences between properties of two classes linked with an annotation (i.e. correspondences between annotated class properties and annotating class ones). This constraint is expressed using the *expression* property. Three *expressionType* can be set: *constraint*, *algebraic* and *discrete*. The property mapping is particularly useful when using the *Is_ case_ of* and *association* (i.e. AnnotationClass) relationships in an a posteriori setting.

The annotation meta-model introduces the relevant resources and concepts required to describe annotations and thus model annotation. It will be set up in next sections to define different kinds of model annotations.

Next step shows how code of annotations editors is generated from the implemented annotation meta-model described above. Part of these editors is automatically generated using the EMF GenModel. Moreover, some part of it is extended to integrate costumed behavior and functionalities.

Finally, the annotation editors are now ready to be exploited for the description of annotation models

#### 5.2.3.2 Implementation of the multi-view analysis tool

In the same manner, the view meta-model is defined and the editors for the construction of a specific analysis view are generated. The view meta-model is implemented as an extension of the Ecore meta-mode. The design model (ideally the enriched design model obtained at the end of the annotation process) and the point of view are described within Ecore and thus conform to the Ecore meta-model. The construction of a specific view and its instances is done through Java transformations.

**The View meta-model**

The view meta-model is built following the model depicted on Figure 3.12. First, the meta-model of Figure 5.6 describes the generic pattern for building a view model and what are the resources (*EClass*) needed for a given analysis. We have used a composition relationship for this purpose.

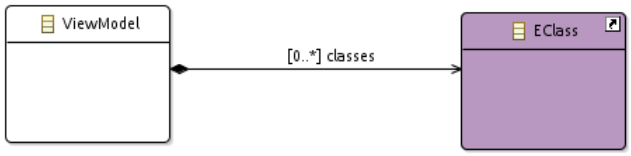



Figure 5.6: The View meta-model to select the required concepts.

The defined meta-model corresponds to the classes appearing on the right and bottom parts of Figure 3.12.

## 5.3 Extension 1. Handling model annotation

### 5.3.1 Creating an annotation project

The first step of model annotation process consists in creating an *annotation project* along with the different models implied in the annotation.

Figure 5.7 shows the selection wizard (File >New >Other) for creating a new *annotation project* (bullet 1), *Ecore Model* (bullet 2) or *EcoreAnnotation Model* (bullet 3).

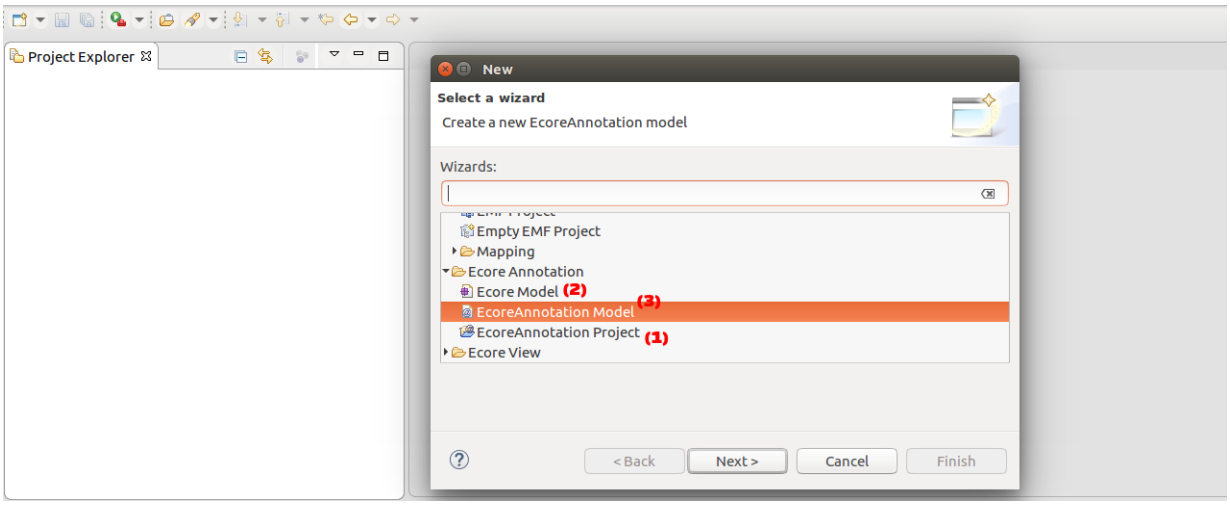



Figure 5.7: Create an annotation project.

Once an *annotation project* is created, four directories are automatically created in the project as depicted in Figure 5.8. An *annotated_models* directory to store the annotation models, a *domain_models* directory to store the design models to be annotated, an *ontologies* directory to store the ontologies models and finally an *enriched_models* directory to store the automatically generated new enriched design models.

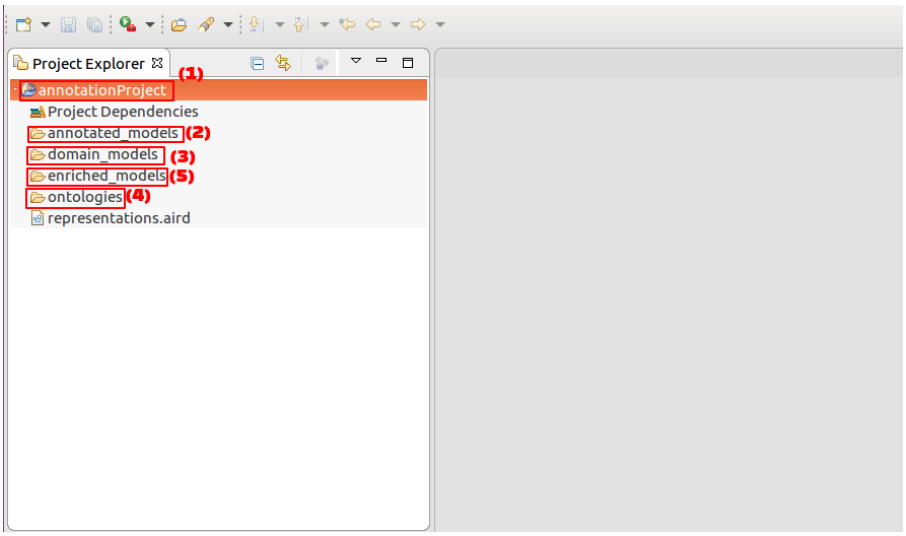



Figure 5.8: Annotation project repertories.

### 5.3.2 The annotation editor

Figure 5.9 depicts the annotation editor (bullet (1)). It allows graphical annotation of design models with explicit references to domain ontologies. First, the annotating classes along with the design model classes to be annotated need to be imported in the editor. This is achieved using the *Import EClass* button (bullet (2)). Then, annotation by *case_of* or by *association* can be performed using respectively the *CaseOf* or the *Annotation Class* buttons (bullet (3)). At the end of the annotation step, a transformation can be trigged to integrate the annotations into a new enriched design model, this is performed using the transformation button (bullet (4)).

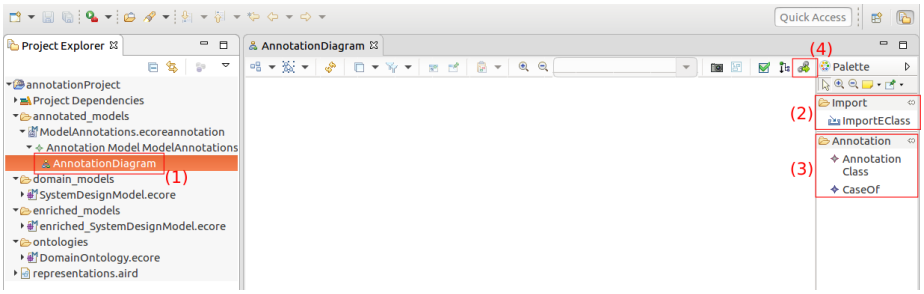



Figure 5.9: Annotation diagram.

### 5.3.3 Annotation by association

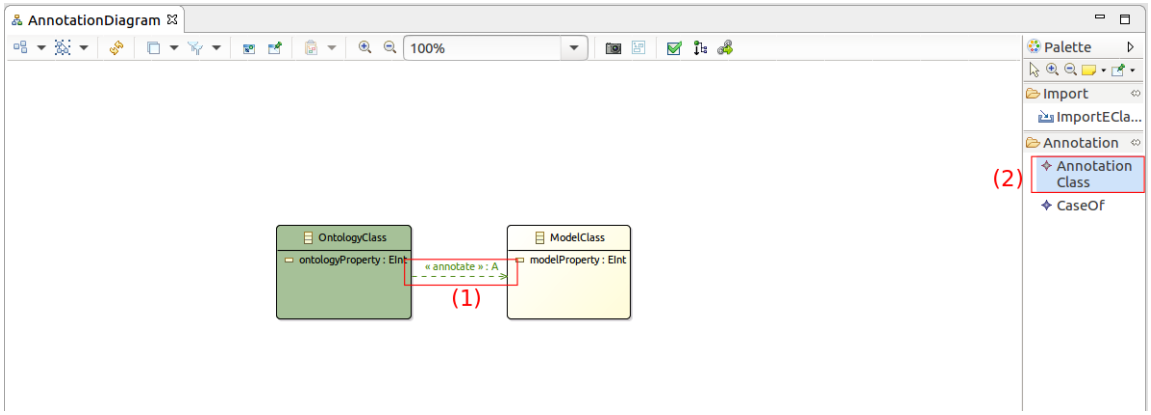



Figure 5.10: An example of annotation: Association.

Figure 5.10 shows an example of annotation that uses the association mechanism to link *OntologyClass* to *ModelClass*. This annotation operation consists in establishing an association relationship (bullet 1) between two concepts (classes): an ontology

concept and a design model one. This type of annotation can be trigged by selecting the *AnnotationClass* button in the *Annotation toolbox* of the annotation editor (bullet 2).

Figure 5.11 shows the annotation by association property editor (accessible by double-click on the *Annotation Class* relation - bullet 1 of Figure 5.10). Properties issued from the two different concepts linked by *Annotation Class* relation (*ontologyProperty* from the *OntologyClass* and *modelProperty* from the design model) are available.

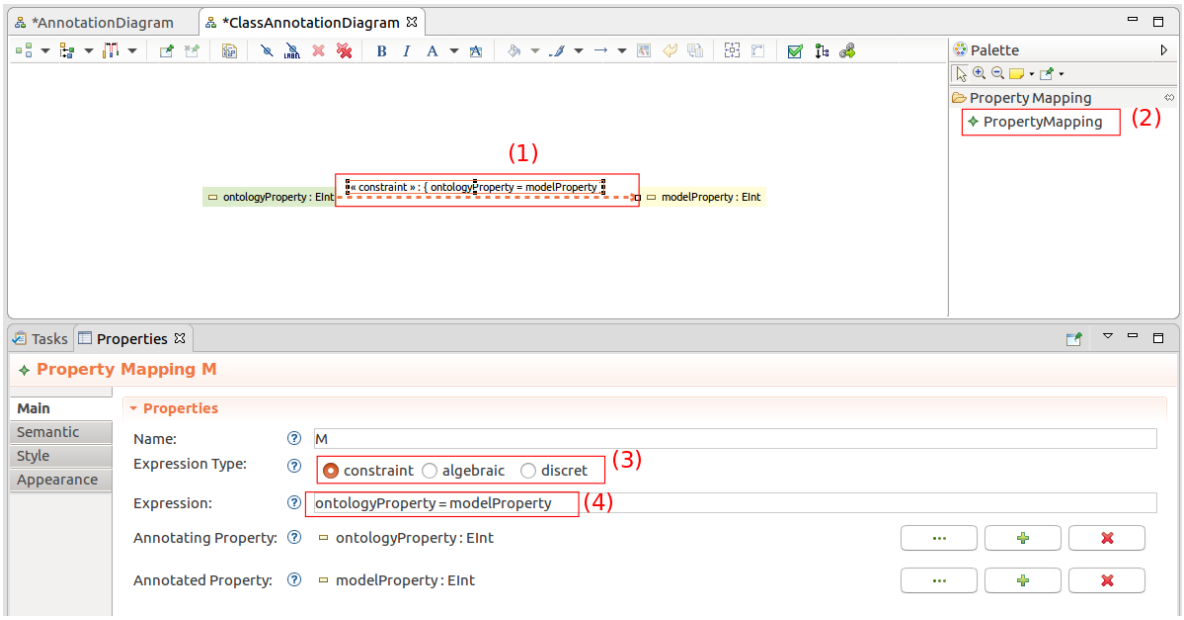



Figure 5.11: Annotation by Association: A property mapping.

This figure shows the particular case where the property *ontologyProperty* is mapped to the property *modelProperty* using a *propertyMapping* relation (bullet 1). An algebraic constraint explicit the exact relationship that exist between the two properties. Thus an algebraic constraint (bullet 3) states that *ontologyProperty* is equal to *modelProperty* (bullet 4).

The property mapping is achieved using the *PropertyMapping* button (bullet 2).
**Remark.** Observe that such a property mapping with algebraic constraint can also be set up in the case of the *case_of* annotation.

### 5.3.4 Annotation by Case_of

The second type of annotation is the annotation by partial inheritance. It is defined by instantiating the *CaseOf* class of the annotation meta-model (Figure 5.5).

Figure 5.12 shows an example of *Case_of* annotation (bullet 1) in the annotation editor. This annotation is achieved by selecting the *CaseOf* annotation in the *Annotation toolbox* of the annotation editor (bullet 2).

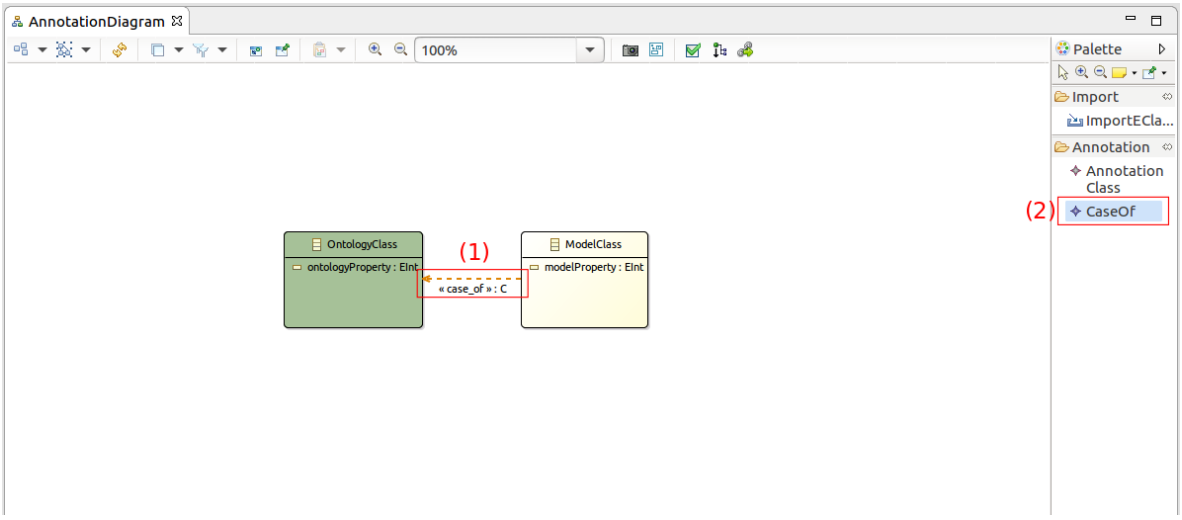



Figure 5.12: An example of annotation: Case_of.

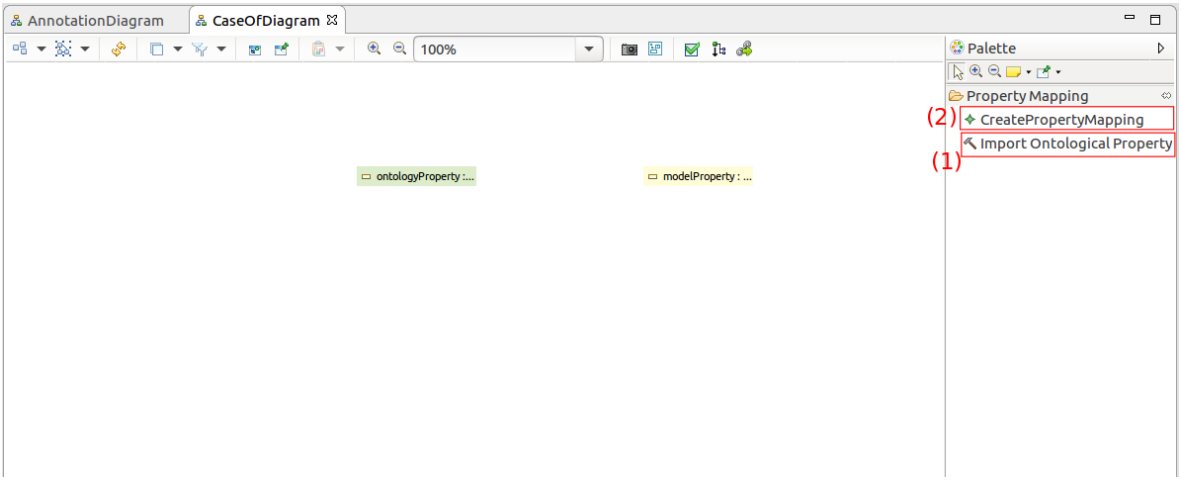



Figure 5.13: Case_of properties editor.

Figure 5.13 shows the property editor of the *Case_of* annotation (appears by double-click on the *Case_of* annotation relationship - bullet 1 of Figure 5.12). Following the *Case_of* partial inheritance principle, the *Import Ontological Property* button (bullet 1) allows the selection of the Ontological properties of the *OntologicalClass* (here *ontologyProperty*) to be inherited and added to the *ModelClass* properties (here

*modelProperty*). Thus, we observe that the *ontologyProperty* ontological property has been inherited and added to *ModelClass* properties using the *Import Ontological Property*. Moreover, the *Create Property Mapping* button (bullet 2) allows to link ontological properties with design model ones by explicitly defining the correspondences that may exists between them. This functionality has been discussed in the *Annotation by association* previous section.

## 5.4 Extension 2. Handling multi-analyses of models

The second step of the proposed methodology consists in handling multi-analyses of design models. This step is depicted on the right hand part of Figure 2.1 presented in chapter 2.

In order to handle such analyses, it is required to supply to the analyzer with the relevant resources for describing the

- analysis to be performed on the design model
- concepts of the design model required by the analysis i.e. input of the analysis
- concepts of the design model or of the defined analysis that shall be returned as a result of the performed analysis i.e. output of the analysis

From a technical point of view, in a model driven engineering setting, the approach is similar to the one deployed for annotating design models.

### 5.4.1 View editor

Figure 5.14 depicts the view editor (bullet (1)). Design model classes can be imported into the view using the *Import* toolbox buttons (bullet (2)). Once the view is built, a view instances generation is needed, this can be performed automatically using the instance generation button (bullet (3)). View instances are then given as input of an analysis tool. The result of an analysis can be caught using the *Get result* button (bullet (4)).

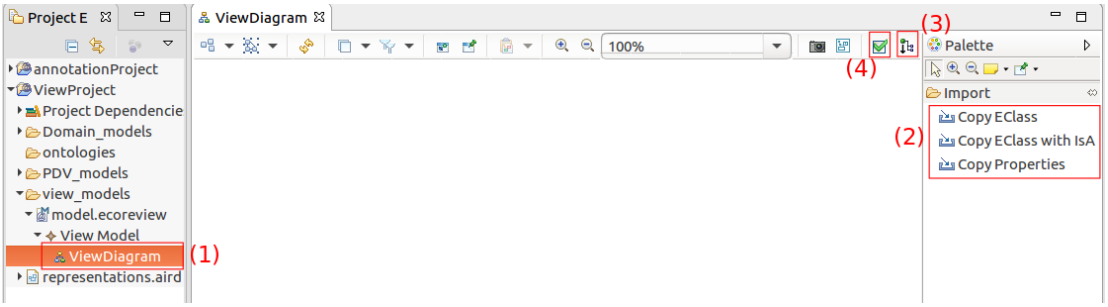



Figure 5.14: Overview of the view editor.

### 5.4.2 Building a view

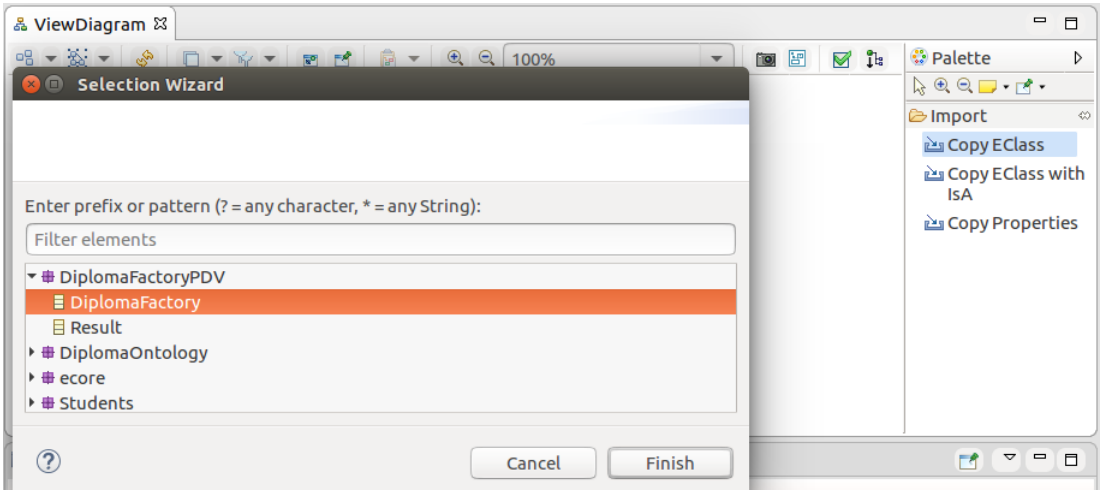



Figure 5.15: Required properties selection in the view editor.

A view model is constructed by importing relevant properties to the specific view from a design model (Figure 5.15). For instance, importing a point of view class using the *CopyEClass* button will automatically import all the design model properties referenced by the *point of view* into the view (directly imported from the design model). Other relevant properties (and not referenced by the point of view) may be needed. Thus, the *Copy EClass with IsA* allows to importation of a Class with all its properties (even the inherited ones) and the *Copy Properties* allows a Case_of inheritance (ie. selecting only some properties of a design model Class to import into the view).

## 5.5 The Event-B context

The Rodin platform has been used for the Event-B setting implementation side. It offers a high level of abstraction and allows the definition of all concepts presented in chapter 4. Thus, no extra tool support is required.

### Rodin tool

The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. The platform is open source, contributes to the Eclipse framework and is further extendable with plugins [58].

The Rodin tool is intended to support construction and verification of Event-B models. The focus is very much on verifying models rather than on verifying programs. No assumptions are made about finiteness of structures and the main verification method is deductive proof; model checking can be used when structures are finite[58] tool is integrated to the Rodin platform for this purpose). Both automatic and interactive proof is provided. The main properties verified of models are well-definedness of expressions, invariant preservation and refinement between models[58].

## 5.6 Conclusion

In this chapter, we detailed the tool-chain implementation of our general framework in the case of MDE and formal setting.

We have developed a Model Driven Engineering tool-chain based on Eclipse technology and more specifically on Eclipse Modeling Framework (EMF) technologies. The Ecore meta-model has been extended in order to integrate the model annotation and multi-view analyses techniques. Graphical syntax and transformation techniques have been provided to support and to ease manipulation models.

The Rodin platform offers a fully supported environment for the deployment of our general framework using Event-B proof and refinement method. Thus, no additional tool support is required.

# Chapter 6

# Validation on embedded systems

In this Chapter, we present the application of our approach on the CORAC-PANDA case study.

This case study deals with an avionic architecture described within static design models. It shows the how our approach improves the quality of this kind of design model when domain knowledge is integrated and taken into account.

We show how the avionic architecture design models described as UML models are enriched and strengthened with new domain knowledge.

In the context of the CORAC-PANDA project, we have studied the particular case of real-time analysis of an avionic system.

## 6.1 Avionic real-time case study

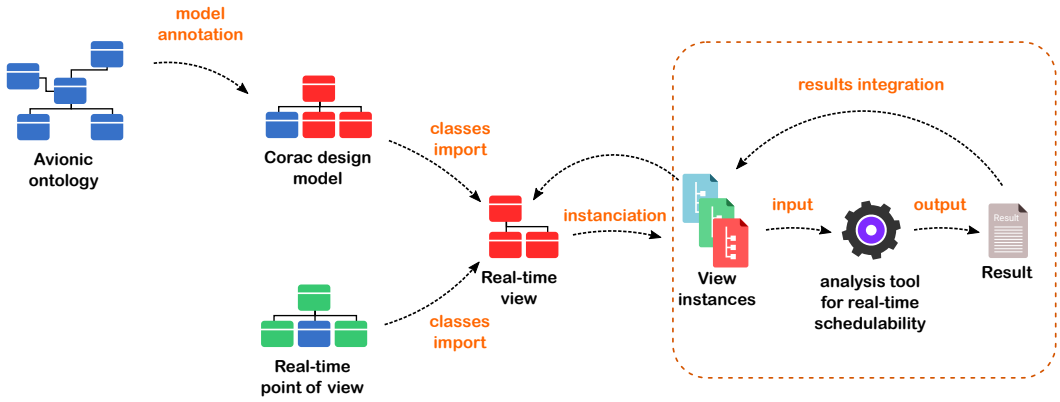



Figure 6.1: Global integrated approach for real-time analysis

Figure 6.1 shows how the global approach defined in chapter 2 can be instantiated for the particular case of real-time analysis.

First, our defined model annotation (section 2.3) is exploited to strengthen the avionic design model with new domain properties. Thus, the initial avionic design model is annotated and enriched by explicit references to avionic domain ontologies.

Second, the *RealTime view* is defined by importing all the required properties from the *RealTime* point of view and the design model. Corresponding *RealTime view* instances are extracted from the instances of the design model in order to create *RealTime view* instances.
The output instances can then be used as input of an *Analysis tool for RealTime Schedulability*. The ouput results of the analysis tool are integrated within the result class of the *RealTime* view.

## 6.2 Annotation of the Avionic real-time meta-model

Next subsections shows the end-to-end application of our model annotation methodology on the Avionic Real-Time case study.

### 6.2.1 Step 1. Domain knowledge formalization

The defined ontology for avionic systems is depicted on Figure 6.2 in a simple class diagram. The avionic platform ontology contains a set of inter-related classes and relevant properties as follows.

- *Thing* is the root concept of all ontologies concepts. It is composed of: *AvionicFunctionalPlatform* for the functional aspects of an avionic platform, *AvionicSoftwarePlatform* for the software aspects of an avionic platform and *AvionicPhysicalPlatform* for the physical aspects of an avionic platform.

- *AvionicFunctionalPlatform* refers to the set of *Function* formalized to perform specific calculi. Each *Function* is characterized with its *name*, *input* and *output* attributes.

- *AvionicSoftwarePlatform* refers to the software aspect of an avionic platform as follows.

    - *Partition* describes the available partitions in a software resource. It is described with properties like *SchedulingAlgorithm* referring the algorithms uses used for distributing resources among different tasks, *intervalStart* and *intervalEnd* which respectively indicate the time where a partition starts and the time it finishes , etc.

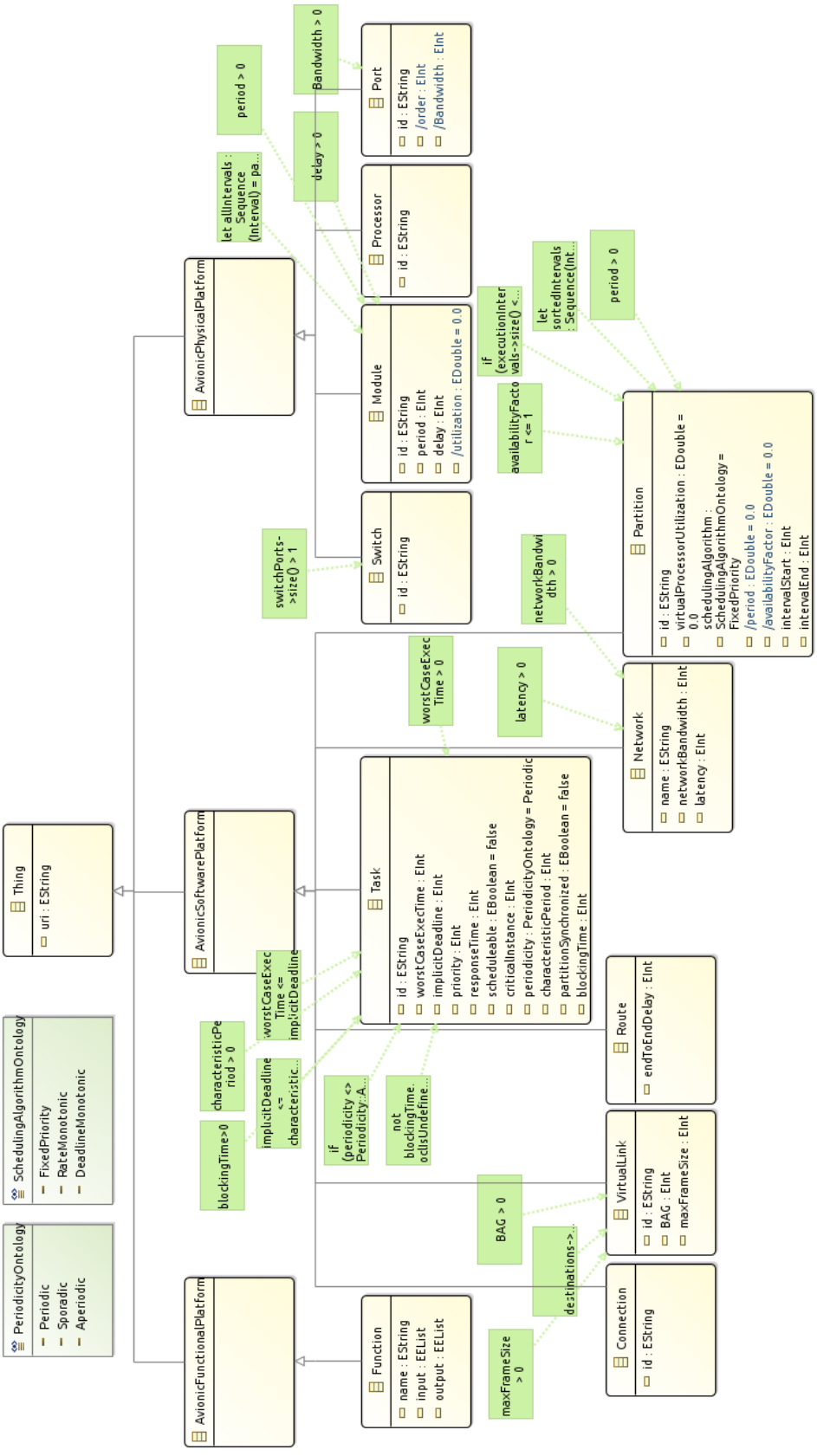



Figure 6.2: Avionic platform ontology.

  - *Task* for the different tasks that may be trigged in a partition. It is characterized with properties like *worstCaseExecTime* for maximum length of time the task could take to be executed, *priority* referring to the priority level that is set for a given task, *responseTime* describing the time elapsed between the dispatch of a task to the time it finishes its job, etc.
  - *Network*, *Route*, *VirtualLink* and *Connection* describe the different software network entities of a platform.

- *AvionicPhysicalPlatform* describes the physical components that are integrated in an avionic platform like *Switch*, *Module*, *Processor*, *Port*, etc. These physical components are characterized with a unique *id*.

Finally, this avionic platform ontology formalize domain properties and constraints. They are depicted in Figure 6.2 as boxes attached to the ontology classes (green boxes).

### 6.2.2 Step 2. Model specification and design

The system's design model is defined according to a given specification. The second step of our approach for model strengthening concerns the definition of avionic platform design model. Figure 6.3 depicts the formalized design model as follows.

- *System* is entry point of the design model. It represents the corresponding avionic platform system and is composed of *HardwareResource*, *SoftwareResource* and *Function* for the functional resource part of the platform.

- *HardwareResource* is of *Module* and *Network* (for the network part of the avionic platform). A *Module* is composed of *Processor* and *EndSystemPort* (ports of a module). The network part of the platform is composed of *Switch* and physical connections *Connection*.

- *SoftwareResource* is composed of *Task* representing the system's tasks that can be executed, *Partition* for the software partitions of the system contained in the modules (*Module*) and *virtualLink* for the virtual links that are defined to map between hardware resources *Module*.

### 6.2.3 Step 3. Model annotation

Figure 6.4 shows the obtained annotated avionic design model. The different concepts are annotated using the defined domain ontology and the defined annotation operators. An extract of this annotation process is depicted. Left, the ontological concepts are used to annotate the right concepts belonging to the avionic design model. *case_ of* and annotation *by association* are set up into the annotation editor. They allow strengthening the avionic design model through Class annotations and properties mapping.

Note that the defined annotations are manual and set by a domain expert. Thus, the guarantee of their soundness is left to the expert achieving the annotation step.

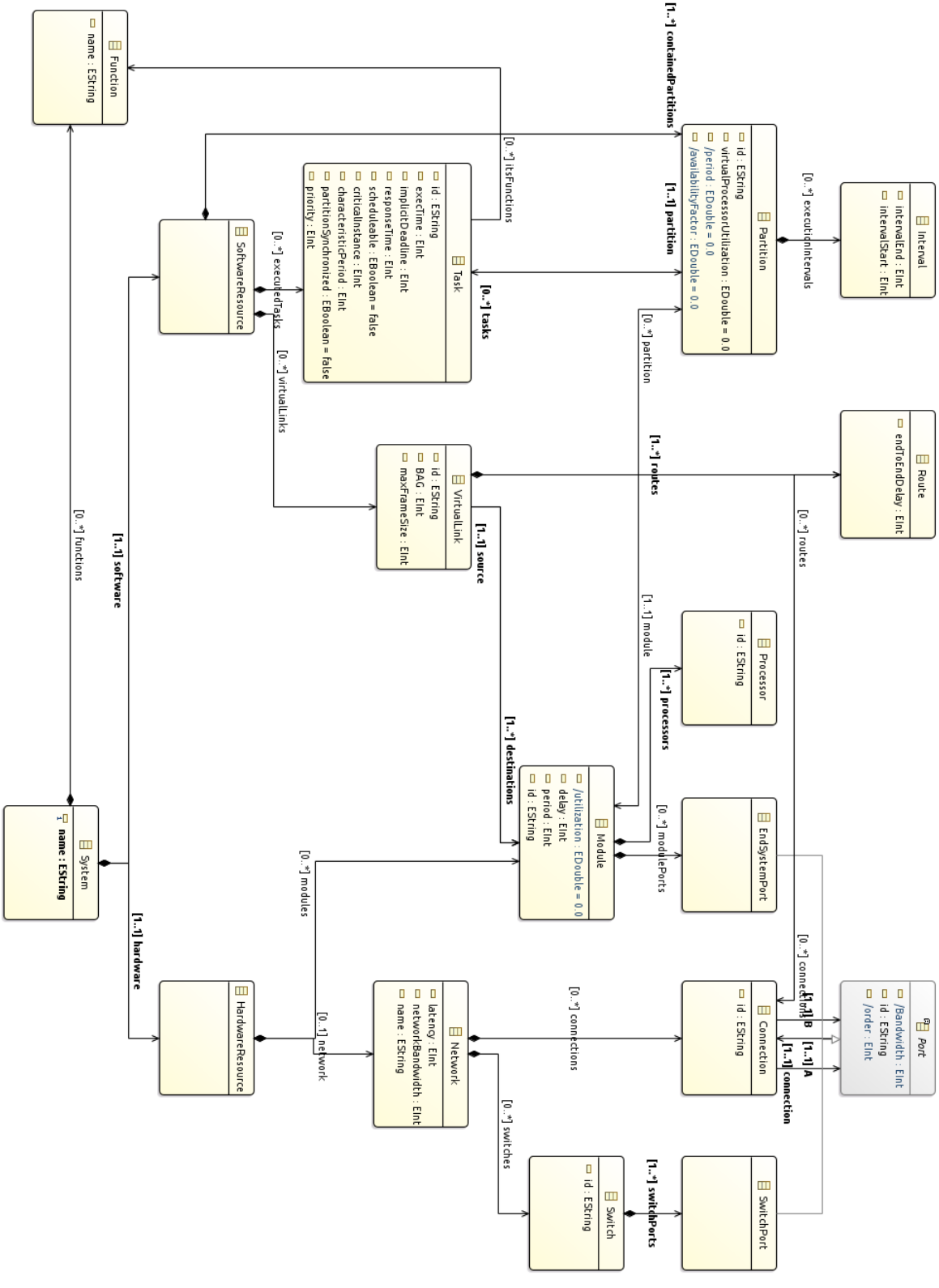



Figure 6.3: Avionic RealTime meta-model.

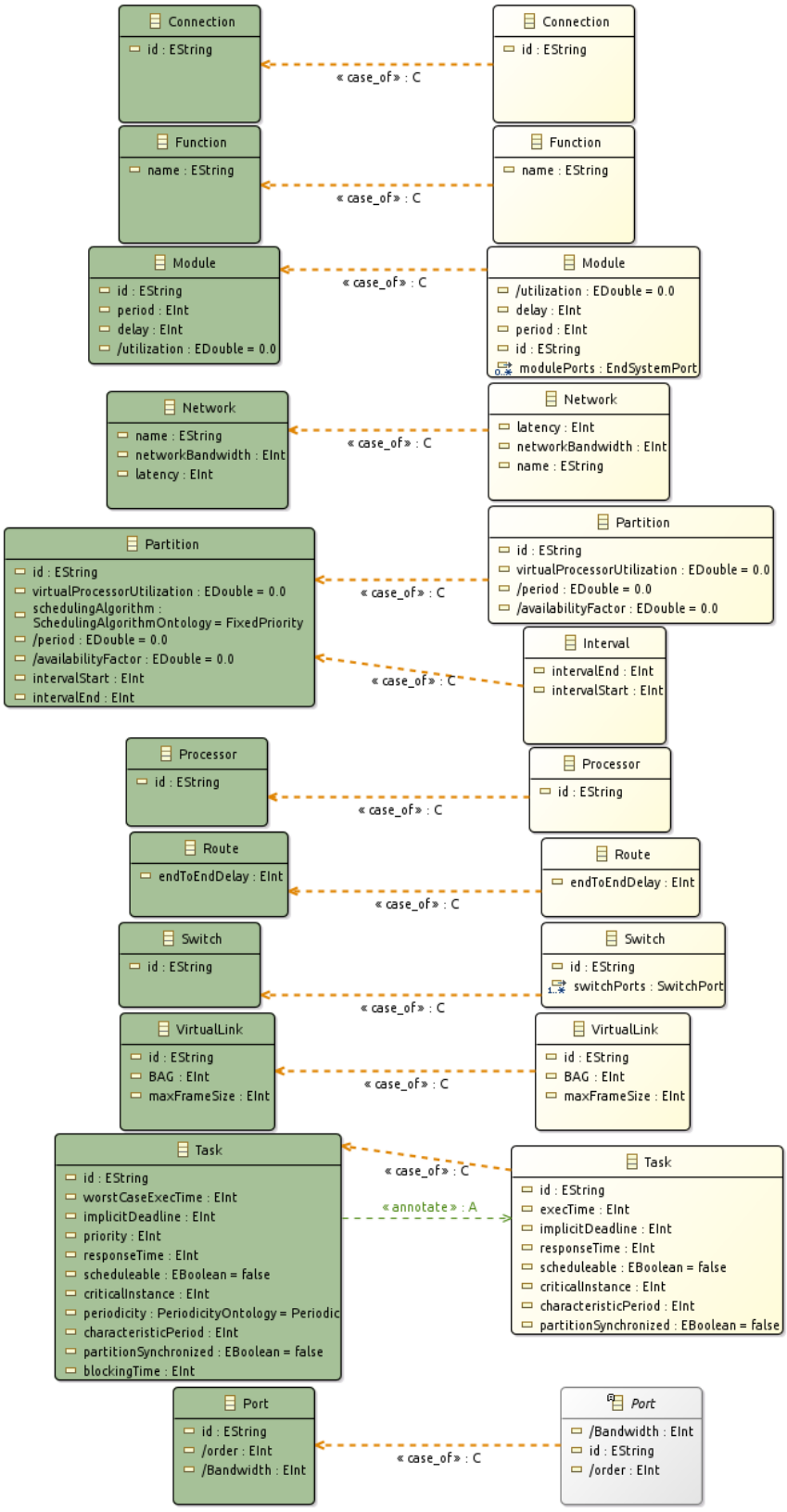



Figure 6.4: An extract of annotation of the Corac meta-model.

### 6.2.4 Avionic real-time enriched meta-model

At the end of the annotation process, Figure 6.5 shows the obtained model. This model is enriched by new properties and constraints borrowed from the domain ontologies. Among the resulting constraints (imported from the ontology through the annotation relationships) two categories can be distinguished.

1. *Expressible constraints* (represented in green color on Figure 6.5) corresponds to the constraints imported from the domain ontology that are expressible and evaluated at the enriched model level since all the properties they are referring to are available at the design model level.

2. *Non evaluable expressible constraints* (represented in orange color on Figure 6.5) are the constraints that are borrowed from the ontology but which can not be valuated. This is due to the fact that the properties or the values (at the instance level) of some of the properties required to evaluate these constraints at the model level are not available (due to the set up annotation relationship).

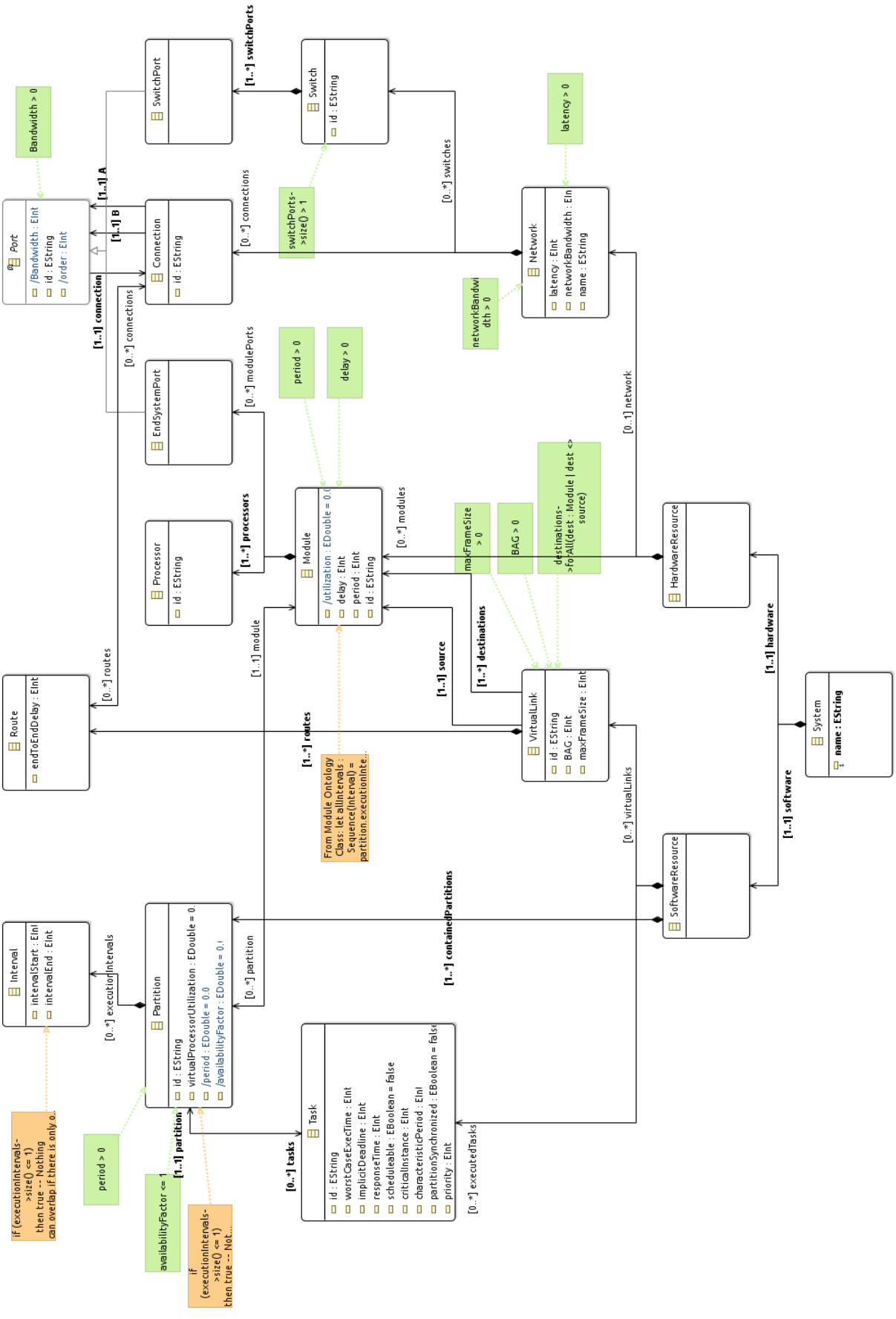



Figure 6.5: Enriched avionic real-time meta-model.

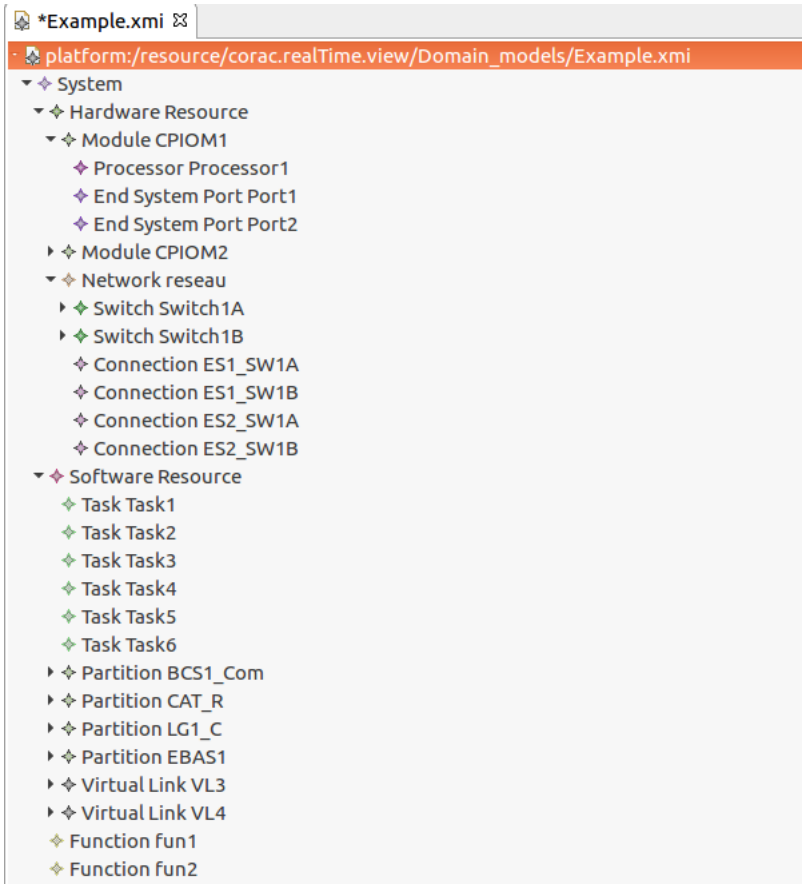



Figure 6.6: An example instance of the enriched Corac metamodel.

When the model is annotated and thus enriched, valid instances can be built. Figure 6.6 shows an example of a valid instance of the enriched CORAC model where instances of hardware, software and functional parts of the new enriched avionic model are instantiated.

The instances of *Module*, *Processor*, *Switch* and *Connection* models classes describing hardware resources are defined. The software resources represented in *Task*, *Partition* and *VirtualLink* instances are also defined. Finally, two instances of *Function fun1* and *fun2* are introduced. They refer to the functional aspect of the avionic platform.

The obtained model together with its instances are now ready for analysis. Next steps show how real-time analysis can be performed on these model and instances.

## 6.3 Multi-view analysis

Next sections show how the multi-view model analysis approach presented in section 3.3 is applied to the strengthened avionic platform design model in order to build a real-time view of this design model and trigger the corresponding real-time analysis.

### 6.3.1 Real-time point of view


Figure 6.7: Real-time point of view.

Figure 6.7 shows the class that describes what a real-time analysis is. This class defines properties like network information, latency, partition, response time, delay, bag, priority, task identification, end to end delay, etc. It also describes the method used to compute a specific real time property. The semantics of these properties are defined in the corresponding domain ontologies.

Notice that not all the design model properties are required for performing a real-time analysis. The identified specific properties are imported for each specific analysis

thanks to the *requiredProperties* attribute of the meta-model defined in section 3.3.

### 6.3.2 Building the avionic real-time view.

In order to build a real-time analysis, it is required to import its description. Figure 6.8 shows the screen capture that defines the importation of the *RealTimePDV* and *Result* resources describing the real-time analysis (bullet 1 of Figure 6.8) thanks to the *Copy EClass* tool (bullet 2 of Figure 6.8) that is defined to import all the required properties (and the view ones) by selecting selecting the corresponding point of view class in the *selection wizard* (bullet 1 of Figure 6.8).

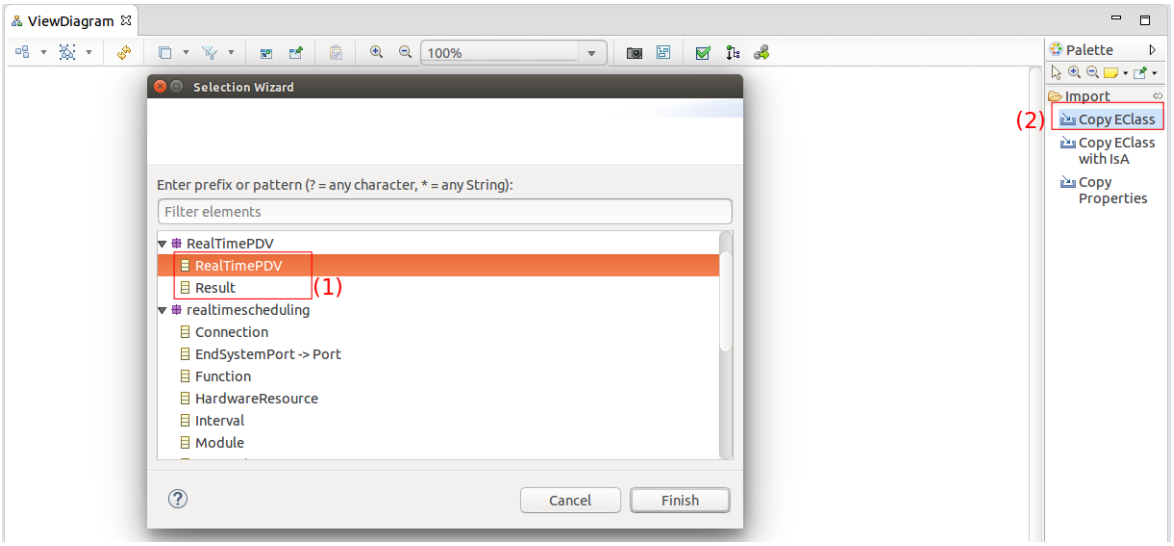



Figure 6.8: Importing *realTimePDV* class.

Once the CORAC model is annotated by references to domain ontologies and the real time analysis (or view) is defined, the two models can be integrated to define the notion of *integrated view* as modeled in the global approach of Figure 3.12.

For the specific case of real time scheduling analysis, Figure 6.9 shows the obtained view after the integration of relevant concepts for performing the real-time analysis. All the required properties referenced by the *RealTimePDV* point of view (Figure 6.7) are imported from the enriched design model (Figure 6.5). These required properties are imported along with the design model classes containing them. The imported classes refer to the hardware (*Connection*, *Module*, etc.) and software aspects (*task*, *Partition*, etc. ) of the avionic platform design model only since the functional properties of the design model are not set as required in the real-time point of view.

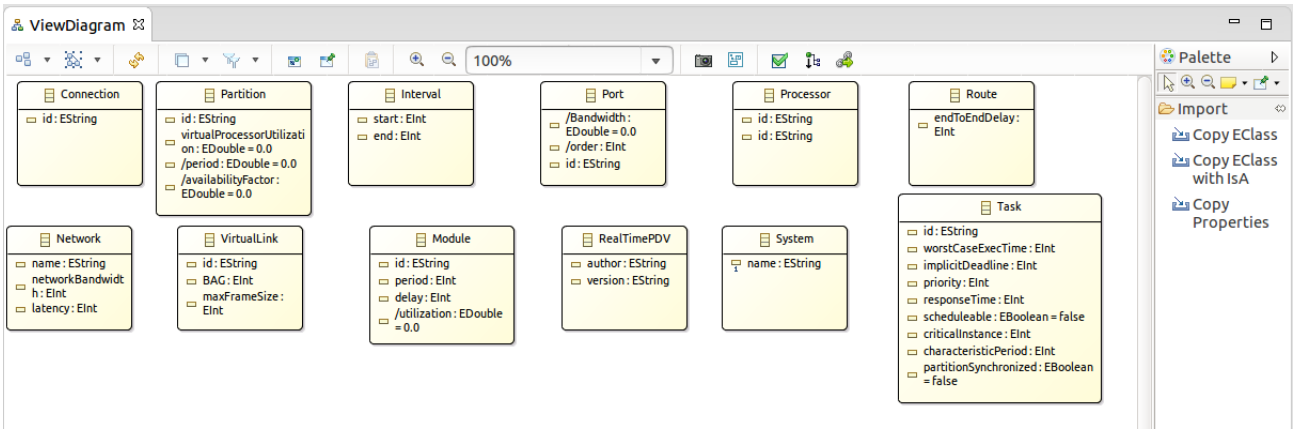



Figure 6.9: View model built with the required properties imported from Corac meta-model

At this stage, all the resources to trigger a real-time analysis are available.

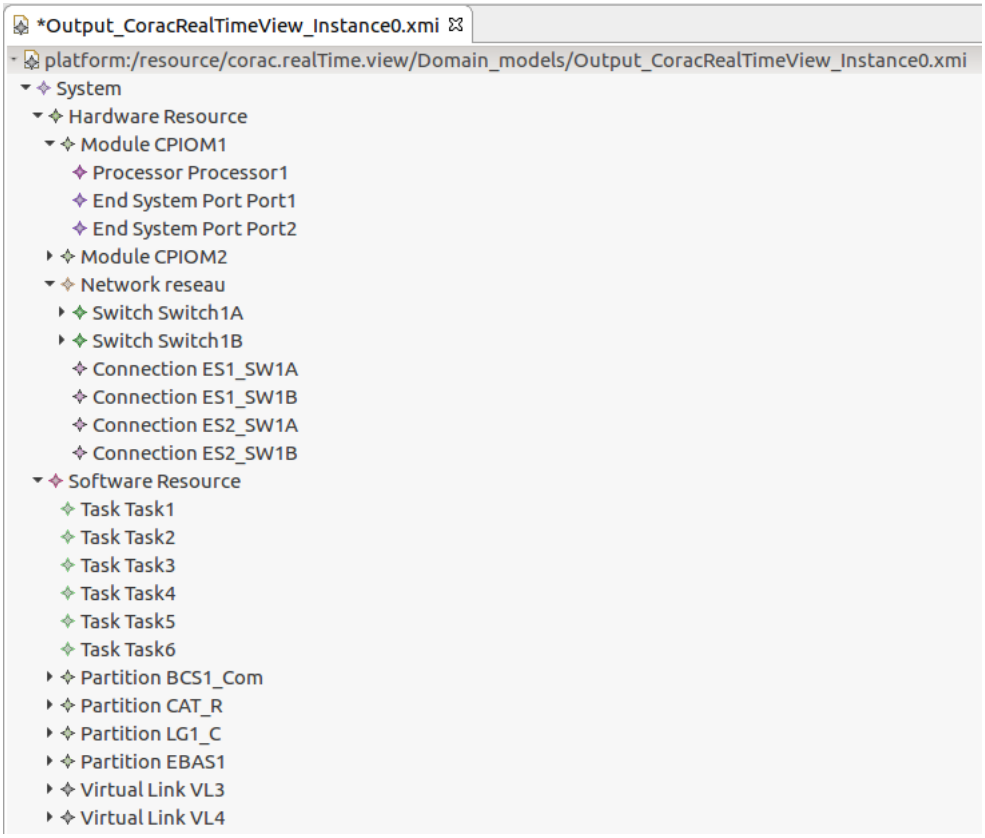



Figure 6.10: Obtained view instance.

In the same manner, the view instances that will be used as input parameters of the real-time analysis resulting from the integration are also obtained. Figure 6.10

shows the view instances for the real-time analysis, they are directly extracted from the enriched design model instances depicted in Figure 6.6. Thus, the enriched design model instance is filtered and the obtained view instance does not integrate *Function* instances since the functional aspect of the avionic platform are not required for this kind of analysis.

### 6.3.3 The exchange process

When the instances are obtained the analysis can be performed. This analysis is triggered with these instances as values for the input parameters of the analysis.

Once the analysis ends, the instance of the *Result* class of Figure 6.7 is returned to the integrated model.

At this level, the system engineer possesses the results of different analysis, he/she is able to take design decisions in order to increase the quality of the defined system models.

## 6.4 Conclusion

In this chapter, we have shown how our approach has been operationally deployed in the engineering domain of embedded systems. We have validated our defined methodology in the case of avionic systems and we particularly studied the real-time analysis of such systems. The design model involved in this case study is directly borrowed from the core model defined by the CORAC-PANDA project consortium.

We have demonstrated, relying on the MDE tool-chain we developed and presented in chapter 5 how a real scale design model is, strengthened using the defined annotation mechanisms (on classes and properties). The new obtained avionic design model is enriched with domain properties and constraints directly imported from the proposed *avionic platform* ontology. This enrichment revealed some design model inconsistencies related to the lack of domain knowledge properties. In fact, some domain constraints could not be integrated in the enriched design model due to the absence of explicit reference to the properties they are related to at design model level. As a second step, this new design model was used to build a Real-Time view. Real-Time view instances have been generated as well. They are used as an exchange format handling all the required properties necessary to trigger a Real-Time analysis on the avionic design model using an external analysis tool.

Finally, our general framework has been validated on several non trivial case studies from the engineering application domain.

# Part III

# Conclusion

# Conclusion and perspectives

## Conclusion

The work presented in this thesis shows the interest of model strengthening and handling of multi-view model analyses by exploiting explicit modeling of domain knowledge through domain domain property expression. A general framework gathering the contributions of our thesis work satisfying ***RG1***, ***RG2*** and ***RG3*** research goals has been defined and set up. The details of its deployment in a Model Driven Engineering setting and in the Event-B formal method based on proof and refinement setting have been shown.

The first part of our work addressed ***RG1*** i.e. the formalization of domain knowledge. We used ontologies for this purpose. We also proposed the core of a unified ontology modeling languages gathering the notations of usual ontology modeling languages. Depending on the chosen modeling language setting, these ontologies are modeled through concepts, relationships between concepts and associated constraints on the one hand and domain axioms and theorems on the other hand.

In the second part, we have proposed a stepwise methodology to fulfill ***RG2***. The defined methodology allows system designers to explicitly handle domain knowledge expressed within ontologies in their design models. The integration of domain knowledge and information during the system specification and design phases allows the developers to handle axioms, hypotheses, theorems or properties mined from the application domain. This requirement is a major concern in system engineering where different standards provide system designers with relevant domain knowledge information but this information is usually not explicitly handled by the design models. To make this information explicit, annotation mechanisms have been defined in both the MDE and formal methods settings. They proved powerful and allowed system designers to map any entity of a design model to another one in an ontology without requiring any change nor modification in the original models. In this way, design

models are separated from the domain model and thus, ontologies and models can evolve separately and asynchronously meeting the separation of concerns objective.

The third contribution of our work fulfilled ***RG3***. It shows the capability to handle model analyses. This idea is not new, but, the novelty of our approach consists in two main improvement. The first one consists in making explicit model analyses through the definition of a descriptive model (point of view) that describes the whole features of a specific model analysis. Ontologies describing the characteristics of model analyses are built and used for this purpose. The second improvement concerns the explicit definition of the required concepts and properties borrowed from a design model to trigger a given model analysis. Indeed, similarly to our process for annotation, when performing a model analysis, our approach keeps trace of the process that allowed a system designer to build its model analysis.

A stepwise methodology for multi-view model analyses has been proposed and deployed for non trivial case studies. This approach makes an extensive use of the model strengthening one. Indeed, a model annotation step is recommended in order to strengthen the quality of the system design models to be analyzed.

Moreover, we have shown in this thesis that ontologies, points of view and design models can be integrated in a unified modeling language. The interest of such integration is semantic alignment where both ontologies, annotations and design models are described in a common shared modeling language.

The work presented in this thesis has been developed as part of the AME Corac-Panda project [2] and ANR-IMPEX project. It has been applied to several case studies issued from engineering domain. Prototypes corresponding to the deployment of our approach in the MDE and Formal methods settings have been developed, and deployed. Experiments with MDE based techniques have been conducted on the particular engineering domain of avionic systems.

# Perspectives

The work presented in this thesis opens several new research directions. Below, we have identified the main issues from both technical and methodological points of view.

From a technical point of view first, the proposed ontology modeling language can be extended and more ontological modeling concepts can be formalized in order to enrich the expressiveness of our approach. Furthermore, the Event-B development

considers ontologies as theories expressed in Event-B contexts. The case where ontologies are defined at an upper level using event-B theories should be studied. Indeed, the use of Event-B theories offers the capability to define built in ontology modeling language concepts and deduction rules. Then, the annotation meta-model should be extended regarding these new formalized ontology concepts and new annotation relationships should be formalized to link new types of ontological concepts.

The annotation mechanisms themselves can be enhanced if automatic annotations techniques are provided. Indeed, inference rules could be added to our developed method and tool in order to detect and deduce automatically possible annotations. Due to the critical aspects of the models we address, an approval from the system designers would still be required before integrating these annotations to the design model.

Moreover, the different models involved in our approach (ontology, point of view, annotation model) are associated with domain constraints and/or algebraic constraints - defined within properties annotation in case of annotation *by partial inheritance* and *by association*. Thus, the defined meta-models for ontology formalization, model annotation and multi-view analysis and associated tool-chain should be extended to define and integrate a language for constraints description along with a procedure to solve and verify them at model and instance levels. The points of view meta-concepts should also be extended and a meta-model for points of view should be defined.

From a methodological point of view first, the case of semantic mismatch, where ontologies, points of view and design models are not described in the same modeling language, needs to be addressed. Semantic alignment shall be studied in the resulting heterogeneous models. Moreover, the engineering application domain we studied relies on modeling languages with classical semantics using closed world assumption (CWA). We are interested in deploying our general framework on other application domains like the semantic web which involves Open World assumption (OWA).

Currently, the developed approach allows a designer to perform a single analysis at a time. We are interested in offering the capability to integrate and ideally compose several model analyses. Allowing such integration will offer different analysis patterns. Furthermore, we are interested in deploying our general framework for other formal methods offering other techniques for the expression, verification and validation of properties different from the ones offered by Event-B.

Finally, our defined general framework deals with static design models only. Its application on dynamic models should be discussed. The formalization of dynamic models including pre and post conditions associated to actions together with their composition at ontological level shall then be studied in the future.

# Bibliography


[1] A free, open-source ontology editor and framework for building intelligent systems, howpublished = http://protege.stanford.edu.

[2] Ame-corac: Avionique modulaire etendue ? conseil pour la recherche aéronautique civile. `http://aerorecherchecorac.com/`.

[3] Eclipse modeling framework. https://www.eclipse.org/modeling/emf/.

[4] Sirius. http ://www.eclipse.org/sirius/.

[5] J. R. Abrial, A. Hoare, and Pierre Chapron. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[6] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[7] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Inf.*, 77(1-2), 2007.

[8] J. Paul Gibson Aït Ameur, Yamine and Dominique Méry. On implicit and explicit semantics: Integration issues in proof-based development of systems. In *ISOLA*, 2014.

[9] Y. Ait-Ameur and H.U. Wiedmer. *General resources*. ISO-IS 13584-20. ISO Genève, 88 pages, 1998.

[10] Yamine Ait-Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. volume 121. Elsevier, 2016.

[11] P.-O. Ribet B. Berthomieu and F. Vernadat. The tool TINA ? construction of abstract state spaces for petri nets and time petri nets. Taylor and Francis, 2004.

[12] Patrick Barlatier and Richard Dapoigny. A type-theoretical approach for ontologies: The case of roles. *Applied Ontology*, 7(3):311–356, 2012.

[13] Ladjel Bellatreche, Guy Pierra, Dung Nguyen Xuan, Dehainsala Hondjack, and Yamine Aït Ameur. An a priori approach for automatic integration of heterogeneous and autonomous databases. In *Database and Expert Systems Applications*. Springer, 2004.

[14] S. Evren P. Bijan. Pellet: An owl dl reasoner. In *International Workshop on Description Logics (DL2004)*, pages 6–8, 2004.

[15] Kalina Bontcheva, Valentin Tablan, Diana Maynard, and Hamish Cunningham. Evolving gate to meet new challenges in language engineering. *NLE*, 10(3-4), 2004.

[16] Kalina Bontcheva, Valentin Tablan, Diana Maynard, and Hamish Cunningham. Evolving GATE to Meet New Challenges in Language Engineering. *Natural Language Engineering*, 10(3/4):349–373, 2004.

[17] N. Boudjlida and H. Panetto. Annotation of enterprise models for interoperability purposes. In *CAISE*, April 2008.

[18] J. P. Bowen, R. W. Butler, D. L. Dill, R. L. Glass, D. Gries, and A. Hall. An invitation to formal methods. volume 29, pages 16–, April 1996.

[19] D. Brickley and R. V. Guha. *RDF vocabulary description language 1.1: RDF schema*. W3C Recommendation 10, 25 February 2014. http://www.w3.org/TR/rdf-schema/.

[20] Jeen Broekstra and Arjohn Kampman. SeRQL: An RDF query and transformation language. In *SWAD Europe Workshop on Semantic Web Storage and Retrieval*, 2004.

[21] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 54–68, London, UK, UK, 2002. Springer-Verlag.

[22] Victorio A Carvalho, João Paulo A Almeida, and Giancarlo Guizzardi. Using reference domain ontologies to define the real-world semantics of domain-specific languages. In *CAISE*, 2014.

[23] Artem Chebotko, Yu Deng, Shiyong Lu, Farshad Fotouhi, and Anthony Aristar. An ontology-based multimedia annotator for the semantic web of language engineering. *Int. J. Semantic Web Inf. Syst.*, 1(1):50–67, 2005.

[24] D. Connolly, I. Horrocks, D. McGuinness, F. Patel-Schneider, and A. Stein. Daml+oil reference description. *World Wide Web Consortium*, 2001.

[25] Hamish Cunningham, Diana Maynard, and Kalina Bontcheva. *Text processing with gate*. Gateway Press CA, 2011.

[26] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. *Text Processing with GATE (Version 6)*. 2011.

[27] Richard Dapoigny and Patrick Barlatier. Modeling ontological structures with type classes in coq. In *Conceptual Structures for STEM Research and Education, 20th International Conference on Conceptual Structures, ICCS 2013, Mumbai, India, January 10-12, 2013. Proceedings*, volume 7735 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2013.

[28] Hondjack Dehainsala, Guy Pierra, and Ladjel Bellatreche. Ontodb: An ontology-based database for data intensive applications. In *Proc. of the 12th Int. Conf. on Database Systems for Advanced Applications (DASFAA'07). LNCS*. Springer, 2007.

[29] Sylvie Després and Sylvie Szulman. Terminae method and integration process for legal ontology building. In Moonis Ali and Richard Dapoigny, editors, *Advances in Applied Artificial Intelligence, 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2006, Annecy, France, June 27-30, 2006, Proceedings*, volume 4031 of *Lecture Notes in Computer Science*, pages 1014–1023. Springer, 2006.

[30] Sylvie Despres and Sylvie Szulman. Terminae method and integration process for legal ontology building. In *Advances in Applied Artificial Intelligence*. Springer, 2006.

[31] Edsger-Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1977.

[32] Chimène Fankam, Yamine Aït-Ameur, and Guy Pierra. Exploitation of ontology languages for both persistence and reasoning purposes - mapping plib, OWL and flight ontology models. In *WEBIST 2007 - Proceedings of the Third International Conference on Web Information Systems and Technologies, Volume WIA, Barcelona, Spain, March 3-6, 2007.*, pages 254–262. INSTICC Press, 2007.

[33] Adam Farquhar, Richard Fikes, and James Rice. The Ontolingua Server: a Tool for Collaborative Ontology Construction. *International Journal of Human Computer Studies (IJHCS)*, 46(6):707–727, 1997.

[34] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *ESOP*.

[35] Anthony Finkelstein, Jeff Kramer, and Michael Goedicke. *Viewpoint oriented software development*. University of London, Imperial College of Science and Technology, Department of Computing, 1991.

[36] Anthony CW Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[37] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[38] Pierra Guy, Aït-Ameur Yamine, and Sardet Eric. ISO (660p), Geneve, 2003.

[39] Volker Haarslev and Ralf Möller. Description of the RACER system and its applications. In *Working Notes of the 2001 International Description Logics Workshop (DL-2001), Stanford, CA, USA, August 1-3, 2001*, volume 49 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[40] Volker Haarslev and Ralf Möller. RACER system description. In *Automated Reasoning, First International Joint Conference, IJCAR 2001, Siena, Italy, June 18-23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Computer Science*, pages 701–706. Springer, 2001.

[41] Kahina Hacid. Handling domain knowledge in formal design models: An ontology based approach. In *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II*, pages 747–751, 2016.

[42] Kahina Hacid. Explicitation de propriétés par annotation de modèles. Technical report, INPT-ENSEEIHT/IRIT, Toulouse, France, Septembre 2014.

[43] Kahina Hacid and Yamine Aït Ameur. Handling domain knowledge in design and analysis of design models. In *Isola Post-Proceedings - 7th International Symposium, ISoLA 2016*.

[44] Kahina Hacid and Yamine Aït Ameur. Strengthening MDE and formal design models by references to domain ontologies. A model annotation based approach. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Proceedings, Part I*.

[45] Kahina Hacid and Yamine Aït Ameur. Annotation of engineering models by references to domain ontologies. In *Model and Data Engineering - 6th International Conference, MEDI 2016, Almería, Spain, September 21-23, 2016, Proceedings*, pages 234–244, 2016.

[46] Siegfried Handschuh and Steffen Staab. CREAM: creating metadata for the semantic web. volume 42, pages 579–598, 2003.

[47] Siegfried Handschuh, Raphael Volz, and Steffen Staab. Annotation for the deep web. *IEEE*, (5), 2003.

[48] Siegfried Handschuh, Raphael Volz, and Steffen Staab. Annotation for the deep web. *IEEE Intelligent Systems*, 18(5):42–48, 2003.

[49] Stephen Harris and Nicholas Gibbins. 3store: Efficient bulk RDF Storage. In *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems (PPP'03)*, pages 1–15, 2003.

[50] Brian Henderson-Sellers. *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Springer Berlin Heidelberg, 2012.

[51] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph, editors. *OWL 2 Web Ontology Language: Primer*. W3C Recommendation, 27 October 2009. http://www.w3.org/TR/owl2-primer/.

[52] Charles-Antony-Richard Hoare. An axiomatic basis for computer programming. *ACM*, 12, 1969.

[53] Gerard Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[54] IMPEX Consortium. Formal models for ontologies, 2015.

[55] ISO. Industrial automation systems and integration - parts library - part 42: Description methodology: Methodology for structuring parts families. ISO ISO13584-42, International Organization for Standardization, Geneva, Switzerland, 1998.

[56] ISO13584-42. Industrial automation systems and integration parts library part 42 : Description methodology : Methodology for structuring parts families. Technical report, International Standards Organization, 1998.

[57] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[58] Michael Jastram and Prof Michael Butler. *Rodin User's Handbook: Covers Rodin V.2.8*. CreateSpace Independent Publishing Platform, USA, 2014.

[59] Stéphane Jean. *OntoQL, an exploitation language for ontology-based databases*. Theses, Université de Poitiers, December 2007.

[60] Stéphane Jean, Guy Pierra, and Yamine Ait-Ameur. Domain Ontologies: A Database-Oriented Analysis. In *Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers*, Lecture Notes in Business Information Processing, pages 238–254. Springer Berlin Heidelberg, 2007.

[61] Stéphane Jean, Guy Pierra, and Yamine Aït Ameur. Domain ontologies: A database-oriented analysis. In Joaquim Filipe, José Cordeiro, and Vitor Pedrosa, editors, *WEBIST (Selected Papers)*, volume 1 of *Lecture Notes in Business Information Processing*. Springer, 2006.

[62] Clifford B. Jones. *Systematic software development using VDM (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[63] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 87–98, New York, NY, USA, 2009. ACM.

[64] John Knight, Jian Xiang, and Kevin Sullivan. A rigorous definition of cyber physical systems. In *Trustworthy Cyber Physical Systems Engineering*. To appear, 2016.

[65] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen. The protege owl plugin: An open development environment for semantic web applications. pages 229–243. Springer, 2004.

[66] Leslie Lamport. The temporal logic of actions. In *ACM Trans. Program. Lang. Syst.*, volume 16, New York, NY, USA, 1994.

[67] Michael Leuschel and Michael J. Butler. Prob: A model checker for B. In *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, pages 855–874, 2003.

[68] Yongxin Liao, Mario Lezoche, Hervé Panetto, Nacer Boudjlida, and Eduardo Rocha Loures. Formal semantic annotations for models interoperability in a plm environment. *arXiv*, 2014.

[69] Yun Lin and Darijus Strasunskas. Ontology-based semantic annotation of process templates for reuse. In *Proc. of the CAiSE*, volume 5. Citeseer, 2005.

[70] Yun Lin, Darijus Strasunskas, Sari Hakkarainen, John Krogstie, and Arne Solvberg. Semantic annotation framework to manage semantic heterogeneity of process models. In *CAISE*, 2006.

[71] Dominique Méry, Rushikesh Sawant, and Anton Tarasyuk. Integrating domain-based features into event-b: A nose gear velocity case study. In *MEDI*, 2015.

[72] Linda Mohand-Oussaïd and Idir Aït-Sadoune. Formal modelling of domain constraints in event-b. In *Model and Data Engineering - 7th International Conference, MEDI 2017, Barcelona, Spain, October 4-6, 2017, Proceedings*, pages 153–166, 2017.

[73] Linda Mohand Oussaïd and Idir Ait-Sadoune. OntoEventB : Un outil pour la modélisation des ontologies dans B Événementiel. In *AFADL 2017*, pages 117–121, Montpellier, France, June 2017.

[74] Boris Motik. KAON2 - scalable reasoning over ontologies with large data sets. *ERCIM News*, 2008(72), 2008.

[75] OMG. Meta Object Facility (MOF) Core Specification Version 2.0, 2006.

[76] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.

[77] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, 2011.

[78] OMG. OMG Object Constraint Language (OCL), Version 2.3.1, January 2012.

[79] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. http://www.w3.org/TR/owl2-overview/.

[80] Zhengxiang Pan and Jeff Heflin. Dldb: Extending relational databases to support semantic web queries. In *In PSSS*, pages 109–113, 2003.

[81] M Jae Park, Ji Hyun Lee, Chun Hee Lee, Jiexi Lin, Olivier Serres, and Chin Wan Chung. An efficient and scalable management of ontology. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA'07)*, volume 4443 of *Lecture Notes in Computer Science*. Springer, 2007.

[82] G. Pierra. Context-explication in conceptual ontologies: the plib approach. In *Proceedings of the 10th ISPE International Conference on Concurrent Engineering (CE 2003), Vol. Enhanced Interoperable Systems*, volume 26, page 2003, 2003.

[83] G. Pierra. Context representation in domain ontologies and its use for semantic integration of data. *Journal on Data Semantics*, 10:174–211, 2008.

[84] G. Pierra and H.U. Wiedmer. Industrial automation systems and integration parts library part 42: methodology for structuring part families. Technical report, Technical Report ISO DIS 13584-42, International Organization for Standardization, 30 May 1996. ISO/TC 184/SC4/WG2, 1996.

[85] Guy Pierra and Eric Sardet. *ISO 13584-32 Industrial automation systems and integration Parts library Part 32: Implementation resources: OntoML: Product ontology markup language*. ISO, 2010.

[86] Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1996.

[87] Douglas C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.

[88] Aditya A. Shah, Aleksandr A. Kerzhner, Dirk Schaefer, and Christiaan J. J. Paredis. *Multi-view Modeling to Support Embedded Systems Engineering in SysML*, pages 580–601. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[89] M. Stocker and M. Smith. Owlgres: A scalable owl reasoner. In *The Sixth International Workshop on OWL: Experiences and Directions*, 2008.

[90] Martin Törngren, Ahsan Qamar, Matthias Biehl, Frederic Loiret, and Jad El-Khoury. Integrating viewpoints in the development of mechatronic products. *Mechatronics*, 24(7):745–762, 2014.

[91] J. Trinkunas and Q. Vasilecas. A graph oriented model for ontology transformation into conceptual data model. *Information Technology and Control*, 36(1A), December 2007.

[92] Martin Verlage. Multi-view modeling of software processes. *Software Process Technology*, pages 123–126, 1994.

[93] Yuxin Wang and Hongyu Li. Adding semantic annotation to uml class diagram. In *ICCASM*, 2010.

[94] Jeannette M. Wing. A specifier's introduction to formal methods. volume 23, pages 8–23, Los Alamitos, CA, USA, September 1990. IEEE Computer Society Press.

[95] Nabila Zouggar, Bruno Vallespir, and David Chen. Semantic enrichment of enterprise models by ontologies-based semantic annotations. In *EDOC*. IEEE, 2008.