A UML profile for feature diagrams: Initiating a model driven engineering approach for software product lines

Thibaut Possompès[1,2], Christophe Dony[2], Marianne Huchard[2], Hervé Rey[1], Chouki Tibermacine[2], and Xavier Vasques[1]

[1] IBM France – PSSC Montpellier
thibaut.possompes,reyherve,xavier.vasques@fr.ibm.com
[2] LIRMM, CNRS and Université de Montpellier 2
possompes,dony,huchard,chouki.tibermacine@lirmm.fr

Abstract. This paper proposes an instrumented approach to integrate feature diagrams with UML models, via UML profiles and a Rational Software Architect plugin. The concrete contribution is the detail of a new UML profile based upon a meta-model synthesising existing feature diagrams semantics, and a Rational Software Architect (RSA) implementation. Our RSA implementation makes possible to link feature diagrams with UML model artefacts. Indeed, it allows traceability between feature models and other different kinds of models (requirement, class diagrams, sequences or activity diagrams, etc.).

Key words: Feature diagrams, UML profile, software product lines, model driven engineering

1 Introduction

Complex IT projects require efficient tools to support IT analysts, architects and developers when they gather client requirements, domain experts advices and implement software.

This is the case in the context of the RIDER project[3] whose purpose is improving energy efficiency of buildings. We think that software product line approach is perfectly appropriate to manage the variations that can be found in our project, and moreover, that feature diagrams will be a great help at gathering specific domains vocabulary and concepts.

In this paper, we describe the creation process to make a UML 2 profile from a state-of-the-art feature meta-model and the corresponding Rational Software Architect plug-in.

[3] The RIDER project ("Research for IT as a Driver of EneRgy efficiency") is led by a consortium of several companies and research laboratories, including IBM and the LIRMM laboratory, interested in improving building energy efficiency by instrumenting it.

This instrumented approach enable integrating and federating feature diagrams with UML models and artefacts (requirements, class diagrams, sequences or activity diagrams, deployment diagrams). It is intended to be used as a part of a general approach for software product lines and for product generation.

This paper is organized as follows. Section 2 presents the feature meta-model and the process required to create the corresponding UML profile. Section 3 describes how the profile has been derived from the meta-model. Section 4 presents how the profile has been implemented in the modelling tool and validated against industrial concerns. Section 5 presents feature diagrams excerpts from our project. Section 6 sum up what has been presented and presents perspectives of further research.

2 Transforming the feature meta-model into a UML profile

This section describes a meta model synthesising the different interesting points that we previously identified after a state-of-the-art. We chose to transform this meta-model into a UML profile to facilitate the integration into UML models. This work is based upon and complete [1] in the way that we produce a meta model that we are using in a very rich industrial project. Our profile implementation is done in Rational Software Architect and we developed a tooling plug-in based upon our UML profile.

2.1 Feature Diagram Meta Model Presentation

Meta-model Description

As depicted in Figure 1, a product line contains features. A product belongs to one product line and is composed of features; features associated to a product must check some constraints, like mutual exclusion or require relations.

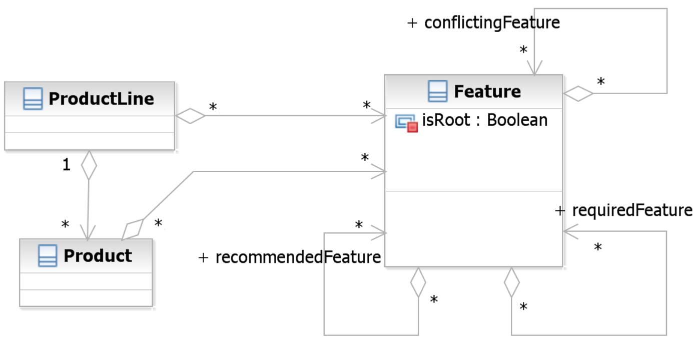


Fig. 1. Feature meta-model excerpt: Product lines, features and products

Mutual exclusion and require relations are modelled by the *conflict* and *require* relationships. The *recommends* relationship advises the user to choose another feature that could be pertinent.

Feature properties (Figure 2) describe either a feature parameter (*e.g.* the bandwidth capacity of a network) or a characteristic chosen by the user (*e.g.* the frequency of automatic backups of a word processing software). The *VariabilityKind* can be: *fixed*, when the property value is fixed throughout all products of the product line; *variable*, when a property value can change, within a product, depending on other features properties; *family-variable*, when the property could vary from product to product accordingly to the selected features; *user-defined*, when the property value can be freely chosen in a given product.

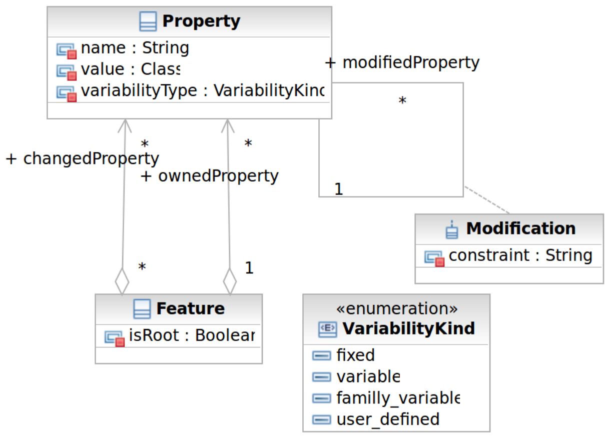


Fig. 2. Feature meta-model excerpt: Feature properties

Figure 3 presents the hierarchy relationships and sub-feature groups. A feature and its sub-features are connected by the *RelationshipGroup* class that contains the cardinalities necessary to restrict the number of sub-features to choose. We highlight special groups like the *OrGroup*, with cardinalities *(0,*)*; *AndGroup*, with cardinalities *(*,*)*; and *XorGroup*, with cardinalities *(0,1)*. A *DirectedBinaryRelationship* links a parent feature and a sub-feature. It can be specialised either by *Enrich*, *Implement*, or *Detail* classes.

In Figure 4, layers and feature sets are linked to the project stakeholders. A stakeholder represents any kind of people; *e.g.* customers, domain experts, IT architect, *etc.*; that can be led to choose features. Feature sets and layers are attached to a specific concern related to the project. A *Layer* represents a view onto the software application. A *FeatureSet* is a *Feature*, and groups features from an arbitrary point of view, *e.g.* for business domain, or representing the features that must be implemented to fulfil a norm.

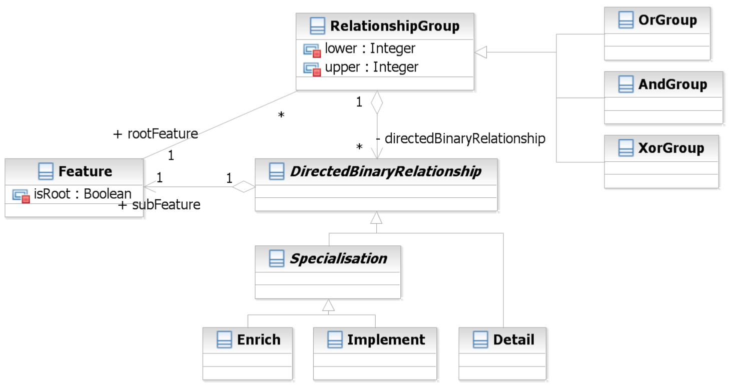


Fig. 3. Feature meta-model excerpt: Groups and hierarchy relationships

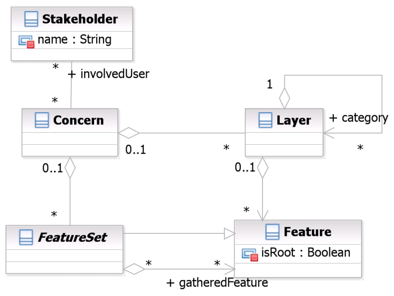


Fig. 4. Feature meta-model excerpt: Groups and hierarchy relationships

Figure 5 depicts constraints on feature sets. A feature set can be either; *mutex*, when only one feature can be selected in the feature set; *None*, when there is no constraint between the features composing the feature set; *All*, when all features or none of them can be selected. A *ConstraintRelation* class is a relation between two feature sets. Hence, one feature set can require another one, two feature sets can mutually require each other, or be mutually exclusive.

2.2 Transformation method

The meta model describes the semantics of the elements used in a feature model. The profile reuses the concepts described in the meta model and integrates them in UML thanks to the profile semantics. Creating a UML profile consists in creating stereotypes that extend UML meta-classes. Stereotypes are meant to add or subtract semantics from the meta-classes they extend. Hence, the first step to create a profile from a meta-model requires to identify which meta-classes or concepts should be transformed into stereotypes, and which UML meta-classes should be extended. This is detailed in the next section. However, the profile

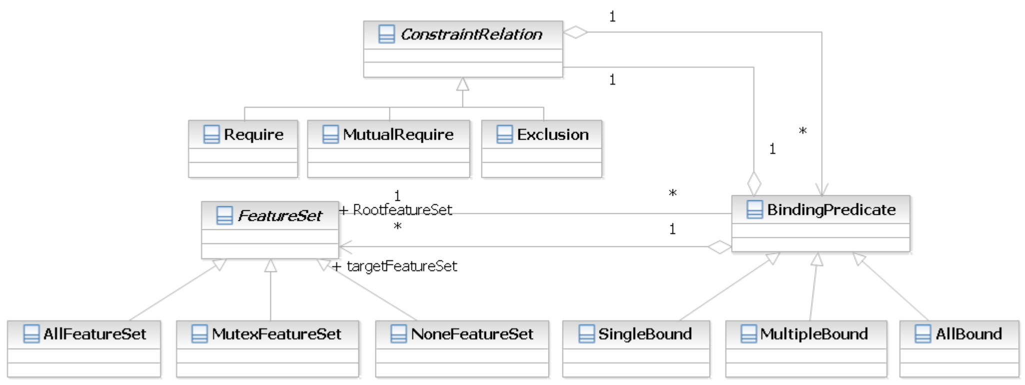


Fig. 5. Feature meta-model excerpt: Constraints on feature sets

could be implemented in very different ways by choosing to extend different UML meta-classes. In our approach, we chose the UML meta-classes that had the closest semantics to our concepts.

3 Profile model

3.1 Meta Classes Extensions and Attributes

Contrary to [2] we choose to base our feature diagram on *Components*. It is the UML concept the closest to what we want to express because they are a high level view of software elements. This first choice will influence the other UML meta-classes selection. For example, the *port* concept allows us to easily group associations to sub-features. This would be impossible if the *feature* stereotype was extended by the *class* UML meta-class. Some associations of the meta-model have been implemented as stereotypes.

- *Feature* stereotype extends the meta-class *Component*.
- *Stakeholder* stereotype extends the *Actor* meta-class because they have a very close semantics.
- *Concern* extends the *Class* meta-class because we will need class attributes to list layers and feature-set references.
- *ModelRelationship* extends a dependency of a feature to any kind of UML artefact. It extends the *Dependency* meta-class.
- *Layer* extends the *Package* meta-class to ease grouping features into a single place.
- *ProductLine* extends *Package* to centralize all software product lines artefacts in a single place.
- *Product* extends *Component* because it represents a set of features which also extend components.

- *Property* stereotype extends the *Port* meta-class to allow us representing them as being artefacts directly attached to a feature. They can be linked to other ports or components. The modification of a property value can be modelled by textual or OCL constraints placed upon the relationship between two properties.
- *RelationshipGroup* extends the *Port* meta-class to represent its belonging to the parent feature. It can be linked only to other features, with directed binary relationships.
- *Modification* extends the *Usage* meta-class.
- *DirectedBinaryRelationship* extends the *Association* meta-class. It is used to link ports to components.
- *BindingPredicate* extends the *Port* meta-class in order to represent the kind of inter feature-set constraint.
- *ConstraintRelation* extends the *Association* meta-class because it links two feature-sets. It can be navigable or not.

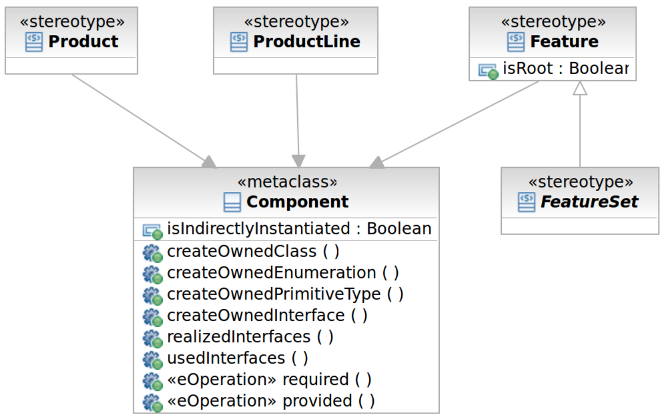


Fig. 6. *Component* UML meta-class extensions

3.2 Resulting Profile

The *ConstraintRelation* stereotype is derived from the meta-model associations *requiredFeature*, and *conflictingFeature*. It is specialised into respectively the stereotypes *MutualRequire* and *Exclusion*. It is not necessary to link their stereotypes to the *feature* stereotype because the *Association* and *Component* UML meta-classes are already connected.

The *BindingPredicate*, *SingleBound*, *MultipleBound* and *AllBound* stereotypes are directly derived from the corresponding meta-classes. There is no need

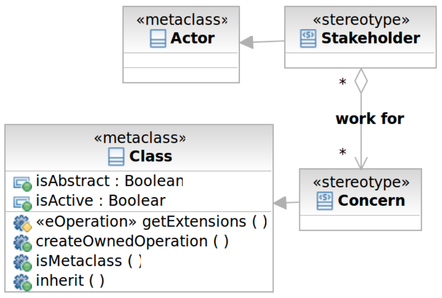


Fig. 7. *Class* UML meta-class extensions

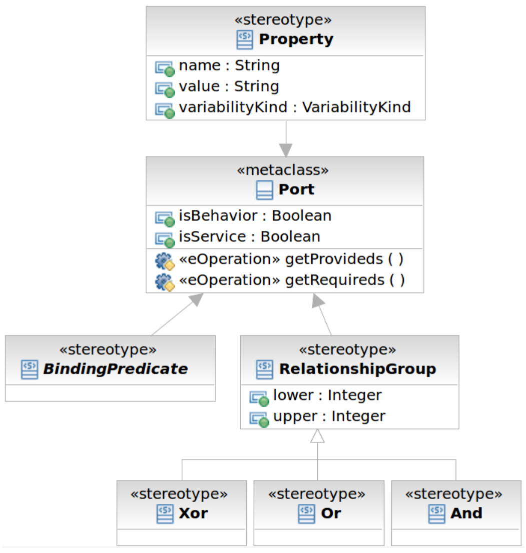


Fig. 8. *Port* UML meta-class extensions

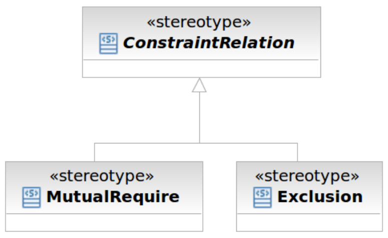


Fig. 9. Constrain relations between features

to link the binding predicate stereotype to the feature-set stereotype because they respectively extend the *Port* and *Component* UML meta-classes that are already linked in the UML meta-model.

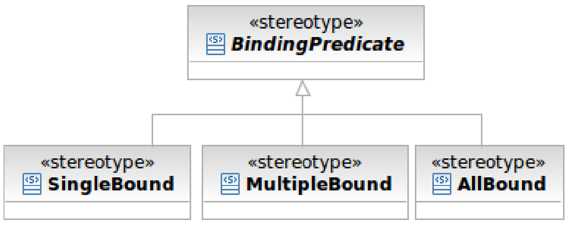


Fig. 10. Binding predicate of feature sets

The *FeatureSet* stereotype and its specialisations are directly derived from the corresponding meta-classes.

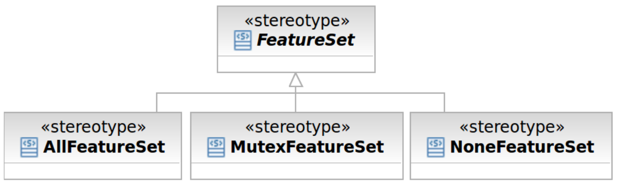


Fig. 11. Feature sets

The *requiredFeature* and *recommendedFeature* aggregations have been modelled the same way that in the meta-model because it is more convenient to have attributes in the *feature* stereotype that list all the required and the recommended features. We could also have modelled these concepts with stereotypes extending the *association* UML meta-class rather than aggregations.

The *Stakeholder*, *Concern*, *Layer*, *FeatureSet*, *Product* and *ProductLine* stereotypes are derived from the corresponding meta-classes. However the *ModellingElement* meta-class has been derived as the *ModelRelationship* stereotype extending the *Dependency* UML meta-class. It represents a dependency between a feature and any UML element, represented by the *Element* UML meta-class.

The *Property* stereotype is not linked to the feature relationship because it extends the *Port* UML meta-class which is already linked to the *Component* UML meta-class. The *Modification* and *ChangedProperty* stereotypes extend the *Usage* meta-class in order to show the impact of one property on another. The *constraint* attribute describe how the changed feature must be impacted.

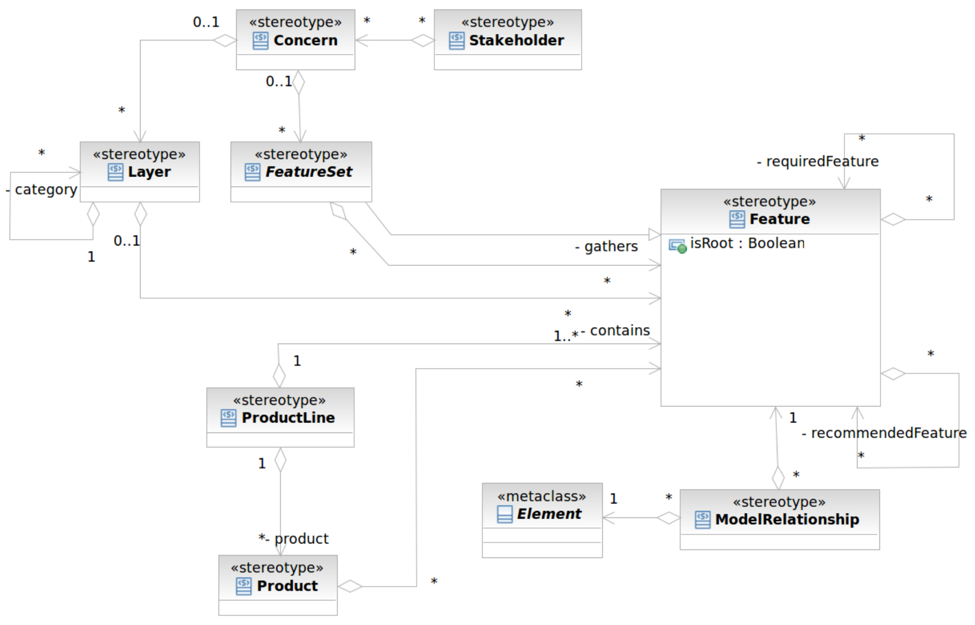
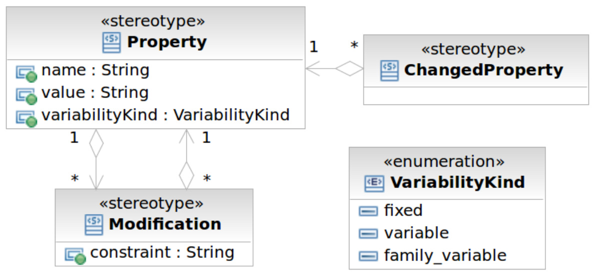


Fig. 12. Feature



«stereotype» Property
name : String
value : String
variabilityKind : VariabilityKind
«stereotype» ChangedProperty
«stereotype» Modification
constraint : String
«enumeration» VariabilityKind
fixed
variable
family_variable
1
*

Fig. 13. Feature properties

The relationships between feature (Figure 14) are directly derived from the meta-model. They are not linked to the *Feature* stereotype because the *Port* and *Association* UML meta-classes are already connected to the *Component* UML meta-class.

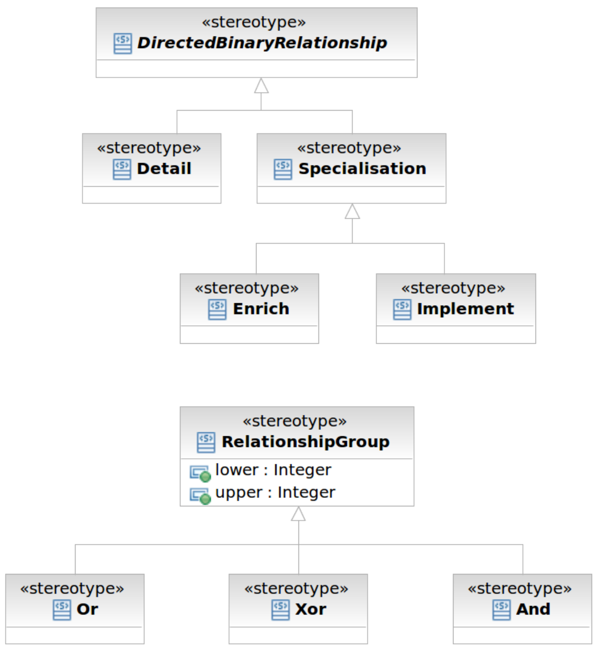


Fig. 14. Feature relationships

4 Implementation and Validation

4.1 Common Profile Implementation Tools

There exist several ways to implement a UML profile and it mostly depends on the tool. We will only discuss how a UML profile can be implemented in *Eclipse*, which is a very widespread platform, and more particularly *IBM Rational Software Architect*, which is based upon *Eclipse*.

The Eclipse Modelling Framework and the Graphical Modelling Framework The *Eclipse Modelling Framework* (EMF) is a modelling framework integrated in the Eclipse platform, which can be used for building tools and applications based upon a structured data model. It integrates the *ECore* meta-model which is equivalent to the Open Management Group's (OMG) Essential Meta-Object Facility (EMOF). This base framework can be used to support any meta-model. Indeed, it provides an implementation of the OMG Unified Modelling Language (UML) 2.x which can be extended thanks to profiles, the standard UML extension mechanism.

We needed a tool to create feature diagrams that could be integrated in a software development life-cycle using UML. The *Graphical Modelling Framework*

(GMF) enable us to develop a graphical editor based upon EMF that fits to our needs.

Rational Software Architect Profile IBM Rational Software Architect is based upon the Rational Modelling Platform which provides a UML modeller, modelling editors, views and tools that are built by using the various services offered by the platform. It also includes several helper components to work with UML models and diagrams. All models managed by the Rational Modelling Platform are instances of EMF models. Hence, using Rational Software Architect allows simplifying tasks like creating a specific plug-in for integrating feature modelling capabilities into standard EMF-based UML models and diagrams.

4.2 Implementation Description

We choose to create a Rational Software Architect plug-in instead-of a standard eclipse-based plug-in. Indeed, it facilitates the tooling source code generation by deriving the code form models describing the plug-in structure [3].

5 Examples

Figure 15 describes the required feature of a building management system. The user must choose one building xml schema, to specify the standard describing its building, and a scenario optimisation set accordingly to his building usage.

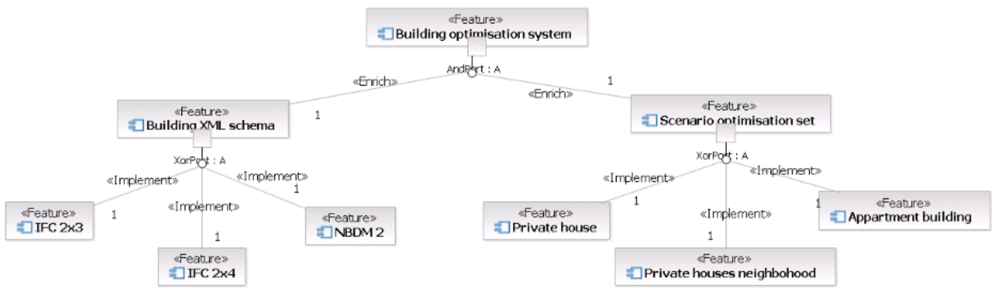


Fig. 15. RIDER feature diagram excerpt

In Figure 16, the business layer contains the features that describe the energetic optimisations that could be done in a building. All features from the business layer are linked to one or several features from the execution model layer. It specify which are the modelling standards able to furnish the required data to each business feature.

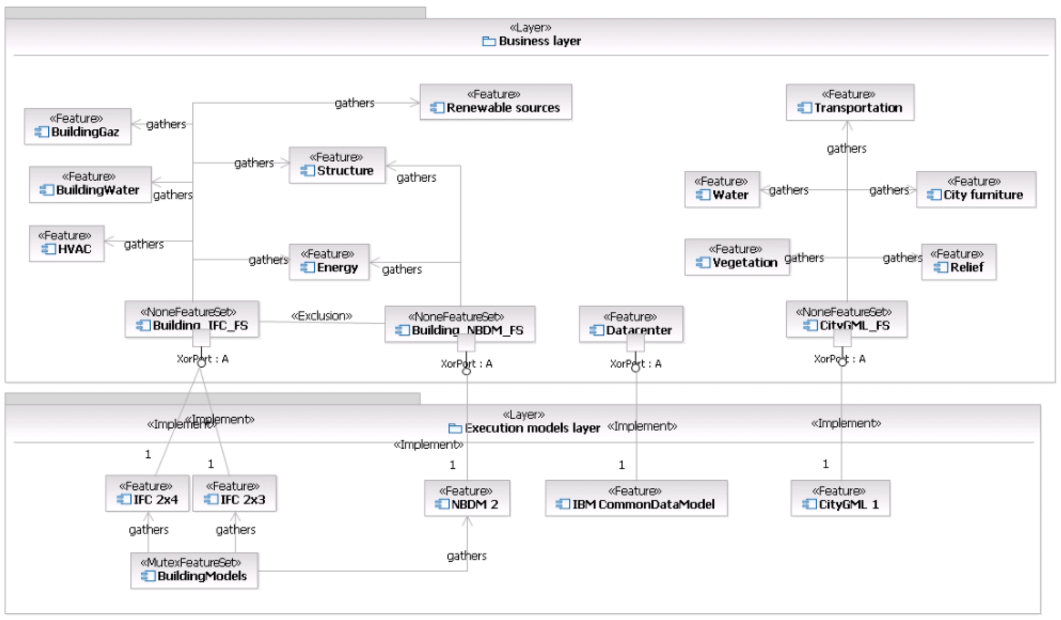


Fig. 16. RIDER feature diagram excerpt

6 Conclusion and Perspectives

We have presented how we implemented a feature profile in UML 2. It is based upon a synthesis of existing work that we enhanced to fit the requirements of the project in which this research is applied. We created a Rational Software Architect plug-in to be able in a later time to easily add functionalities required by our industrial project.

References

1. Asikainen, T., Mannisto, T., Soininen, T.: A unified conceptual foundation for feature modelling. In: Proceedings of SPLC '06, IEEE Computer Society (2006) 31–40
2. Clauss, M.: Untersuchung der Modellierung von Variabilität in UML. Technische Universität Dresden, Diplomarbeit (2001)
3. : Authoring UML profiles using rational software architect and rational software modeler. http://www.ibm.com/developerworks/rational/library/05/0906_dusko/