SysML and UML Models Usage in Knowledge Based MDA Process

Audrius Lopata[1], Martas Ambraziunas[1], IlonaVeitaite[1], Saulius Masteika[1], Rimantas Butleris[2]

[1]*Kaunas Faculty of Humanities, Vilnius University,*
Muitines St. 8, LT-44280 Kaunas, Lithuania
[2]*Centre of Information Systems Design Technologies, Faculty of Informatics,*
Kaunas University of Technology,
rimantas.butleris@ktu.lt

Abstract—The article presents Enterprise Model (EM) generation process from SysML models of four types (Use Case, Activity, Block Definition and Requirements) as well as Knowledge Based MDA (Model-Driven Architecture) tool's prototype which is implementing the defined algorithms. Defined SysML models parsing algorithms use the recursive data processing approach due to complex UML models structure. The algorithms description is presented as Activity diagrams and explained in tables, where are depicted basics steps and actors as well as the output results. Knowledge Based MDA tool's prototype currently is capable of processing Use Case and Activity models. Proposed prototype is implemented using three layers (GUI, Logic and Data) architecture. The detailed architecture is presented and described in this article using class diagrams. The prototype is implemented using .Net framework and C# programming language.

Index Terms—UML, SysML, MDA, enterprise model, knowledge based IS engineering.

I. INTRODUCTION

The UML (Unified Modelling Language) is a modelling language that first appeared in 1997. UML has become one of the most widely used modelling languages in industry. The usage of UML varies through application domain: from embedded systems to information system. Users of business field also use UML for modelling their business domains as well as processes. The current version of UML is UML 2.5 (beta), released in October 2012 [1].

The role of UML in software development has become more significant since the appearance of model-driven architecture (MDA), which has set the UML in the central place of software development. Despite the advanced usage of UML in a wholesome model-driven software development (such as MDA) many software development projects actually use UML in various approaches. Still, UML as a modelling language is mostly used in the context of conventional model-driven development, where models are used by programmers as the basis for implementation. This stage is mostly manual and based on empirics. Besides, the style and strictness of using the UML in modelling also diverge. This difference depends on many factors such as analysts' or designers' experience, time constraints and user requirements.

SysML is based on UML and involves modelling blocks instead of modelling classes, accordingly providing a vocabulary that is more suitable for Systems Engineering. SysML reuses a subset of UML and defines its own extensions. Therefore, SysML includes nine diagrams instead of the thirteen UML diagrams. Both SysML and UML languages are based on the same Meta Meta-Model, (OMG Meta Object Facility (MOF)) [2]. SysML is considered both a subset and an extension of UML 2.0. As a subset, UML diagrams considered too specific for software (Objects and Deployment diagrams) or redundant (Communication and Time Diagrams) were not included in SysML. Some diagrams are derived from UML without significant changes (Sequence, State-Machine, Use Case, and Package Diagrams), some are derived with changes (Activity, Block Definition, Internal Block Diagrams) and there are two specific diagrams (Requirements and Parametric Diagrams). SysML is compatible with UML, which can facilitate the integration of the disciplines of Software and System Engineering [3].

Enterprise meta–model is formally defined enterprise model structure, which consists of a formalized enterprise model in line with the general principles of control theory. Enterprise model is the main source of the necessary knowledge of the particular problem domain for IS engineering and IS re–engineering processes [4]–[6].

SysML and UML modeling languages are used in the Knowledge Based MDA IS development method [4]. Knowledge Based MDA (KB-MDA) is the Information Systems (IS) development method that combines the best practice of Knowledge Based IS engineering as well as the main principles of MDA based IS development processes. The main steps of KB-MDA method are as follows:

1. To acquire problem domain knowledge to Computation Independent Model (CIM) using SysML models;
2. To transform knowledge from CIM [7], [8], [5] to Enterprise Model (EM). In particular case CIM consist of SysML models. Transformation process includes SysML models data consistency validation procedure;
3. To validate EM against Enterprise Meta-Model (EMM);
4. To transform knowledge from EM to Platform Independent Model (PIM);
5. To generate Platform Specific Model (PSM) from PIM;

6. To generate programming code from PSM.

KB-MDA method specifies and ensures two types of knowledge transformation directions: from CIM to EM and from EM to PIM. This is the main difference from standard MDA approach that uses direct [9]–[11] transformation from CIM to PIM. The knowledge acquired using SysML models (Use Case, Activity, Block Definition, Requirements) is transformed to Enterprise Model (EM) in order to validate those according to Enterprise Meta-Model (EMM) rules. The second procedure is performed in order to validate knowledge that is necessary for generation of IS design stage UML models. Basically EM participates as a hidden layer responsible for enterprise knowledge validation against formal criteria specified in EMM. According to the ideal scenario, system's analyst will use SysML and UML models, because these modelling languages (especially UML) are "de facto" standards of IS development. The additional validation of MDA based IS development process becomes real challenge when there is growing market of mobile applications that should be created for multiple platforms, but provide almost identical functionality. Basically, the MDA approach is very well suited for this purpose [12].

In this article SysML models transformation to Enterprise model algorithms are provided as well as architecture of Knowledge Based MDA tool's prototype that implements part of described transformations.

II. ENTERPRISE MODEL GENERATION USING SYSML MODELS ALGORITHM

SysML models usually have a complex hierarchical structure. In order to parse these models, recursive data acquisition method should be used. The main idea is to select a particular model element, to create EM representation of it, then perform analysis, in case this element has child elements (related elements) and iterate through child list (create child objects in parental element).

Each child element should be treated as parent element as well. This process should be repeated while there will not be any unprocessed elements left. This approach is used by KB-MDA based information system for parsing SysML models (Use Case model, Block definition model, Activity model, Requirements model) and generating UML models (Class, Sequence, Package) from EM. SysML models are parsed in a strictly defined order [6]. The first step is parsing of the Use Case model is. This model is used to specify high level functional requirements and activities that internal structure is specified using Activity models. These two models specify dynamic knowledge structure of a particular problem domain. The third step is parsing of Block Definition model that specifies static problem domain knowledge. The final step is parsing of Requirements model (it specifies non-functional requirements that are associated with EM elements that were created using three previous models). UML models generation doesn't have such a strict order. These models can be generated by demand of the developer or system architect. Detailed SysML models parsing algorithms are described in the chapters bellow.

III. ENTERPRISE MODEL GENERATION FROM USE CASE MODEL

Use Case models are used to capture and represent system's behavioral information and basic scenarios. The main elements of this model are as follows: *Actor*, *Use Case*, *Package, Include* and *Extension Relationships*. These elements are used as basic entities in order to complete EM objects. The main idea of this process is to iterate through all *Actors* in Use Case model and using their data to create particular EM objects. Each *Actor* is related to one or more *Use Cases* and *Use Cases* can be related by *Include* or *Extend* relationships types. These relationships represent *Business Rules* or generalization relationships. Parsing algorithm of Use Case model is presented in Fig. 1 and the results are described in Table I.
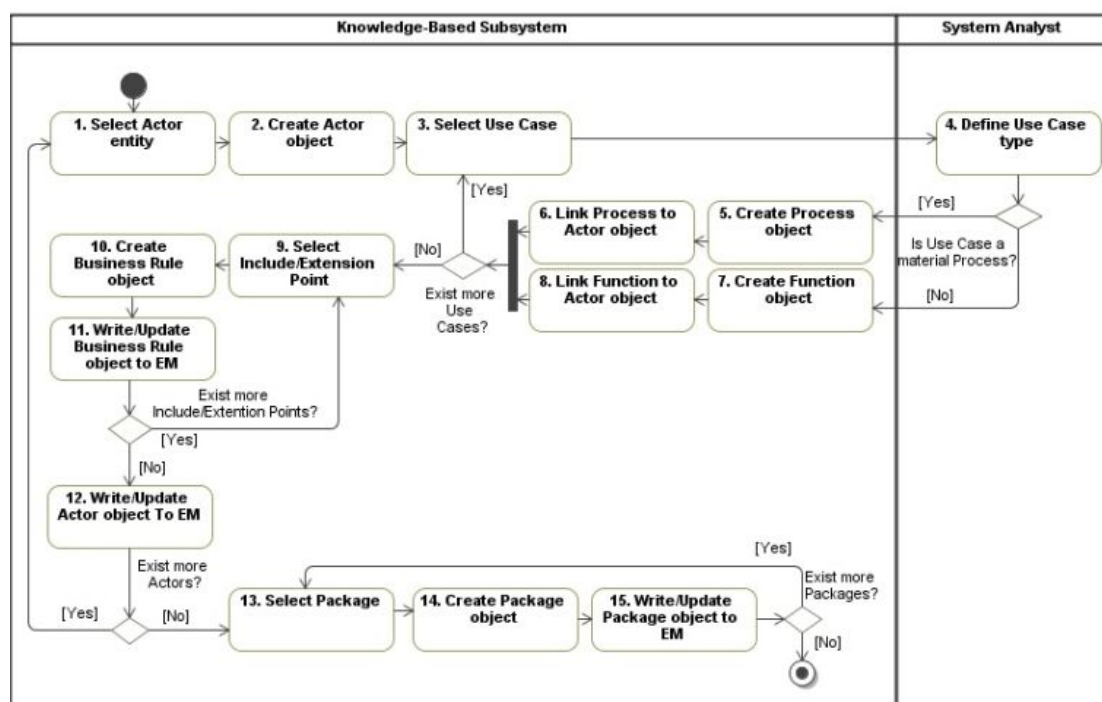

Fig. 1. The main steps of EM objects generation from SysML Use Case model.

Input (objects from Use Case model) and Output (objects created to Enterprise model) elements are presented in the table below.

TABLE I. UML USE CASE MODEL ELEMENTS.

Use Case model object	EM object	Mappings description
Actor	Actor	*Actor* is an entity that defines problem domain "executors". The Use Case models allow creating initial EM *Actors* that contain only basic information (e.g. name).
Use Case	Function, Process	*Use Case* is the UCM's entity that contains dynamic problem domain information (Activities). Depending on particular *Use Case* type, it can be converted to *Function* or *Process* in EM. More detailed information about problem domain Activities are acquired using Activity model.
Include relationship	Business Rule	Include relationship defines *Business Rule* that specifies what predecessors are needed for particular *Use Case* to be performed.
Extension relationship		Extension relationship specifies *Business Rules* that can extend particular *Use Case*. This object (relationship) can be compared to generalization relationship in Class model.
Package	Package	*Package* is general grouping element through various UML diagrams. In Use Case model it is used to group *Actors* and *Use Cases*, depending on functional areas.

Parsing of Use Case model is the first step in EM generation process. After this step is completed initial EM objects are created. These objects are empty templates at the moment. Additional data will be acquired parsing other SysML models (Activity, Block Definition, Requirements). The next EM generation step is Activity model parsing. This model provides information about Flow dynamics as well as basic information about problem domain elements.

IV. ENTERPRISE MODEL GENERATION FROM ACTIVITY MODEL

Activity model is used to capture system's behaviour knowledge and define interactions among system's objects (including rules that define these interactions). Activity model is heavily related to Use Case model and represents (in most cases) the same knowledge but in a more detailed manner. The main elements of Activity model are as follows: *Swimlane*, *Action*, *Decision Point* and *Merge Point*, *Object Flow* and *Control Flow*. Each *Swimlane* represents particular EM A*ctor*. Activity model's *Actions* can represent EM *Processes* or *Functions*. System analyst must decide which object (*Process* or *Function*) should be created for particular *Action*. *Control Flows* define the sequence in which *Actions* are performed. Any input or output *Object Flows* are represented as *Informational* or *Material Flows* in EM. EM *Business Rule*s are created based on the conditions stored in *Decision* and *Merge Points*. Activity model parsing algorithm is presented in Fig. 2 and described in the Table II.

In the Table II a detailed description of Input (objects from Activity model) and Output (objects created/updated to Enterprise model) elements is presented.

As it can be seen from Fig. 2 the process is iterative and is performed while there aren't any processed *Swimlane* or

related elements to EM left. Parsing of Activity model is the second step of EM generation process. After this step is performed EM objects are updated by additional information as well as new objects are created (such as *Event*). The next EM generation step is parsing of Block Definition model.
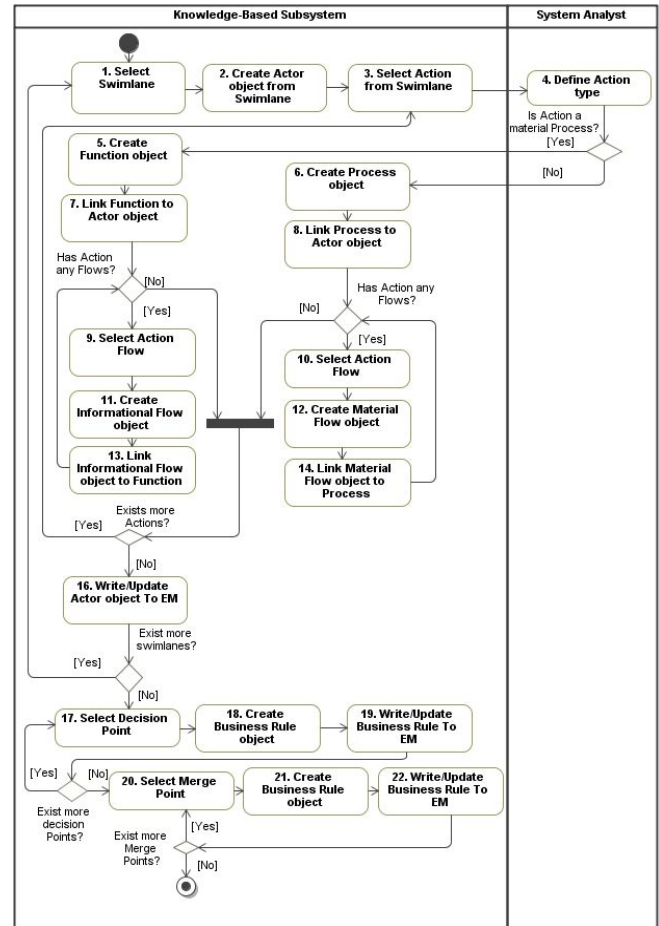

Fig. 2. The main steps of EM objects generation from SysML Activity model.

TABLE II. UML ACTIVITY MODEL ELEMENTS.

Activity model object	EM object	Mappings description
Swimlane	Actor	*Swimlane* represents *Actor* object in EM. Each *Swimlane* is used to define new *Actor* in EM. *Swimlane* contains *Actions* that represent various *Actor's* activities and data objects. Data objects are shared among *Swimlanes*.
Action	Function, Process, Informational Activity, Event	*Action* represents *Function, Process or Information Activity* objects in EM. *Action* can have incoming or leaving *data objects*. *Event Action* is a special type of *Action*. It defines trigger, which is performed by external system or environment.
Control flow		*Control Flow* defines the sequence in which *Actions* are performed.
Object flow	Material Flow, Informational Flow	*Object Flow* defines *Object Flows* among actions and *Swimlanes*. Each *Action* can receive object, perform some changes with it (or create new object) and pass it to the next *Action*.
Decision/ Merge point	Business rule	*Decision/Merge point* defines *Business Rules*. Using these points *Actions* routing data will be handled.

This model provides information about static system's elements.

V. ENTERPRISE MODEL GENERATION FROM BLOCK DEFINITION MODEL

Block Definition model is used to capture system's static information including structure and hierarchy. This model is processed after Use Case and Activity models have been processed. The main purpose of Block Definition model is to provide attributes for EM objects. The main elements of Block Definition model are as follows: *Block, Actor*, *Operation*, *Operation Parameter, Constraint and Property.* Block Definition model parsing algorithm is presented in the Fig. 3 and the results are described in the Table III.
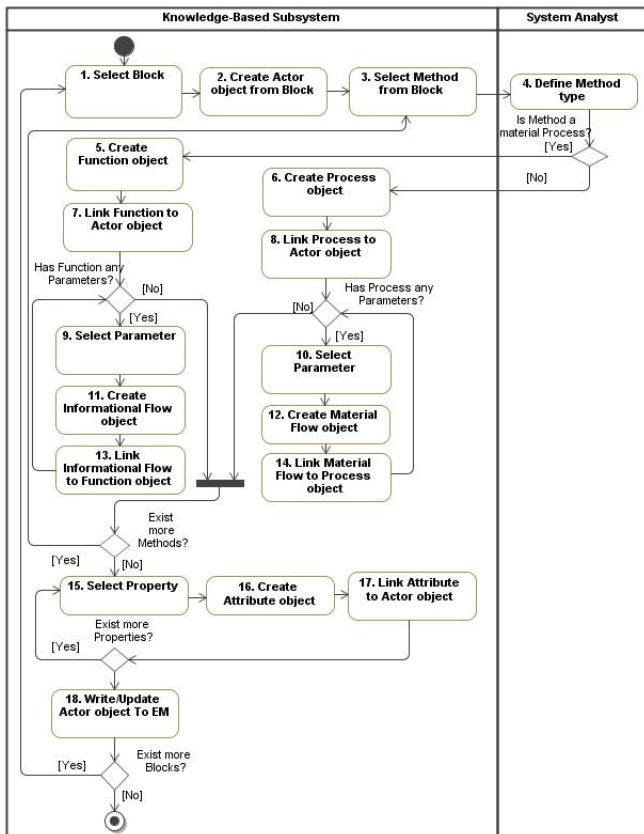

Fig. 3. The main steps of EM objects generation from SysML Block Definition model.

In the table below a detailed description of Input (objects from Block Definition model) and Output (objects created/updated to Enterprise model) elements is presented.

TABLE III. UML BLOCK DEFINITION MODEL ELEMENTS.

Block Definition model object	EM object	Mapping description
Block	Actor	*Block* represents *Actor* entity in EM. This element contains information about a particular element of a system. *Block* can have a hierarchical structure and it represents attributes or generalization relationship as well.
Operation	Function, Process	O*perations* are routines that specify behavior of the represented *Function* or *Process* in EM. Systems analyst must decide if the selected *Operation* represents *Function* or *Process*.

Block Definition model object	EM object	Mapping description
Operation Parameter	Material Flow, Informational Flow	*Operation Parameters* are input or output objects that are used by internal activity.
Property	Attribute	*Property* is a *Block* element that specifies static *Block* information. *Property* has name and value (or reference to object type). It is mapped to *Attribute* object in EM.
Constraint	Business Rule	*Constraints* define *Block* restrictions and are represented in EM as *Business Rules*.

Block Definition model parsing is an iterative process. It is repeated while all *Blocks* and *Blocks'* elements are processed. Parsing of Block Definition model is the third step in EM generation process. After this step is performed EM objects are completed with static problem domain knowledge. The last EM generation step is the parsing of Requirements model. This model provides knowledge about non-functional system requirements.

VI. ENTERPRISE MODEL GENERATION FROM REQUIREMENTS MODEL

Requirements model provides knowledge about system's non-functional requirements. The main purpose of this model is to assign requirement to EM objects (*Actors, Functions*, *and Processes*). The main elements of Requirements model are: *Requirement* and *Attribute*. Requirements model parsing algorithm is presented in the Fig. 4 and the results are described in the Table IV.
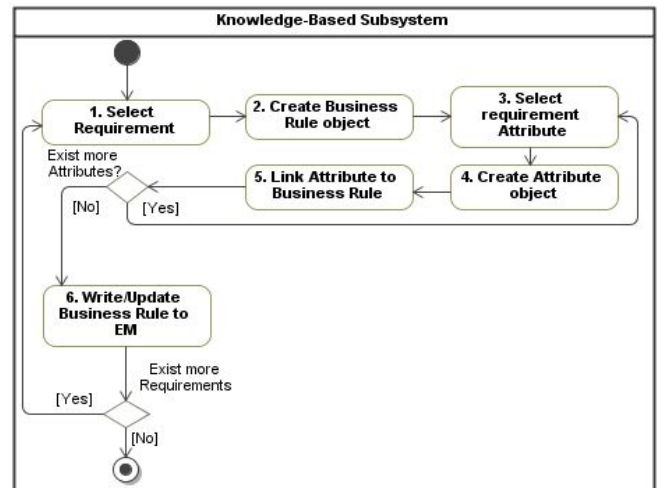

Fig. 4. The main steps of EM objects generation from SysML Requirements model.

In the table below a detailed description of Input (objects from Requirements model) and Output (objects created/updated to Enterprise model) elements is presented.

TABLE IV. SYSML REQUIREMENTS MODEL OBJECTS.

Requirements model object	EM object	Transition description
Requirement	Business rule	*Requirement* is an object that provides information about system's non-functional requirements. This object is mapped to *Business Rule* in EM.

Requirements model object	EM object	Transition description
		Business Rules define performance requirements to *Processes* and *Functions*.
Attribute	Attribute	*Attribute* is an object that is used to store static information about the requirement. *Attribute* has name and value.

Requirements model parsing is an iterative process. It is repeated while all *Requirements* are processed. Parsing of Requirements model is the last step in EM generation process. After this step is performed EM objects are linked with non-functional system requirements.

VII. PROTOTYPE

Described UML models parsing algorithms are partly implemented in Knowledge Based MDA tool's prototype. The main goal of such tool is to be integrated into particular modelling tool as plug-in [13] and provide Knowledge Based MDA functionality. The prototype is capable of parsing and processing common to UML and SysML Use Case and Activity diagrams. These diagrams should be created using particular CASE modelling tool [14] and transformed to in XMI [15] format. Architecture of this prototype is presented in the chapters below. The architecture of KB-MDA tool consists of ten separate modules connected by dependencies. All modules are assigned to one of the three categories: UI Layer, Logic Layer, and Data Layer (Fig. 5).
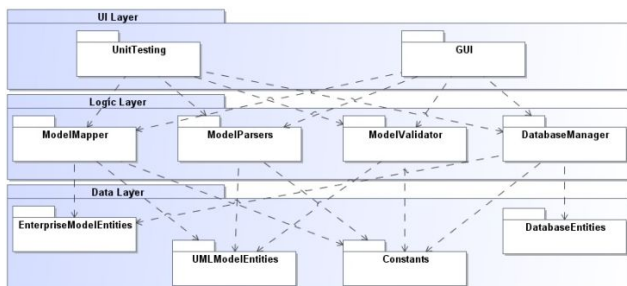

Fig. 5. Knowledge-Based MDA tool prototype's architecture.

There can be used different approaches of creating MDA based tools [16]. The architecture is created in such way that most elements can be reused as separate libraries. Short description of modules functionalities are provided below and a full description is provided in the next chapters.
UI Layer:
− *GUI* module contains user interface elements such as forms and dialogs. This module handles the interaction with user and user's data input/output.
− *Unit Testing* module contains unit testing classes and it is used for automated testing of application *Logic* elements and *Entities*.
Logic Layer:
− *ModelParsers* module contains objects for UML models parsing from XMI format and creating *UMLModelsEntities* in prototype.
− *ModelValidator* module is responsible for various UML models validations (e.g. consistency).
− *ModelMapper* module contains objects that are used for *UMLModelEntities* transformation to enterprise model

(EM) Entities.
− *Database Manager* module handles *EnterpriseModelEntities* transformation to DatabaseEntities and saving/loading to enterprise model database.
Data Layer:
− *UML Model Entities* are created by parsing XMI file. These entities are used for model validation as well as data input for EM Entities generation.
− *EM Entities* are used for EM analysis and validation. These are the building blocks of EM.
− *Database Entities* contains entities used by ORM for DB related actions.
− Constants module contains static information for the application such as Constants, Enums, Messages, etc.

A. GUI and Unit Testing

GUI module is responsible for handling system's interaction with user. User interface is organized using dialogs. All dialogs can be divided into two main categories: input dialogs and output dialogs. The list of currently existing dialogs is presented below:
− MainForm. This form is applications starting point. Using provided Menu user navigates from this form to other dialogs.
− ParseUMLModelsDialog. Dialog used for selecting selection CASE tool XMI file and parsing it. This form contains statusBar that shows model parsing progress.
− ValidateModelDialog. Using this dialog user selects models for validation. Currently there is implemented models consistency validation. In consistency validation user selects Main/Mirror models pair and prototype is checking relationships among models elements.
− LoadModelsDialog. Dialog shows UML Models that are parsed from XMI file. Models are presented in ordered list.
− ViewModelElementsDialog. Dialog shows all single UML model Elements. Model elements are showed in ordered list providing element name and type information.
− ViewValidationResultsDialog. Dialog is used for showing validation results. Dialog represents information about elements from Main model and the existing/missing elements from Mirror model.
− MapModelDialog. Map dialog represents mapping information i.e. how UML model elements are converted to EM elements.
UnitTesting module contains classes and routines that are used for testing purposes only. The unit testing is used to automate testing procedures and test application functionalities such as model parsing, validation, and mapping, entities saving and loading to DB. The main purpose of unit testing is to maintain application consistency and integrity during development.

B. Parser

This module contains objects and routines used for XMI file parsing. Each UML model type (Use Case, Activity, and Class) should have its own parser. All parsers implement *IParser* interface and are created using *Factory* Design Pattern. *Parser Factory* is used to return particular *Parser*

depending on model type. Each *Parser* can have various methods for parsing different models as models internal structure is different. The parsers are using Response/Request data input output patterns. Each parser responses should contain not only information about parsed UML model's objects but as well status about parsing process and error messages if some exceptions were encountered. Module elements are presented in Fig. 6 and the results are described in Table V.
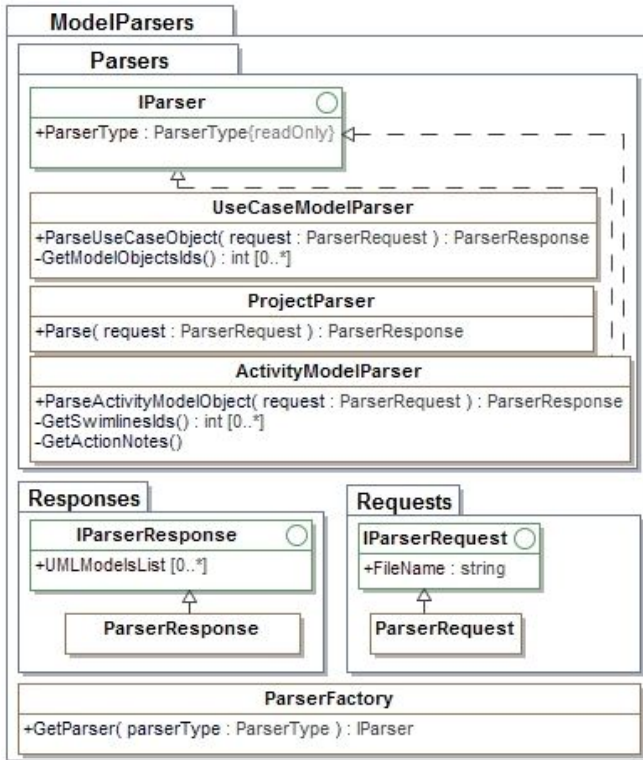

Fig. 6. ModelParsers module internal elements.

TABLE V. MODEL PARSERS MODULE INTERNAL ELEMENTS.

Object	Description
IParserResponse	Interface for parser responses (Status and Error message properties).
IParserRequest	Interface for parser responses (File name property).
IParser	Interface for parsers. Provides basic method for parsing.
ParserResponse	Class used to return parsing result data.
ParserRequest	Class used to provide parsing input data to parser.
ParserFactory	Class used to create particular parser depending parser type.
ActivityModelParser	Class that contains methods used for Activity model parsing.
UseCaseModelParser	Class that contains methods used for UseCase model parsing.
ProjectParser	Class that contains methods used for parsing all models contained in XMI file.

ProjectParser is used to parse whole XMI file. All newly introduced parsers should implement *IParser* interface and to be included into *ParserFactory* and *ProjectParser* routines.

C. Validator

This module contains objects and routines used for UML models validation. Currently in prototype is implemented model consistency validation for Use Case and Activity models. Consistency validation as input uses two models *Main* model and *Mirror* model. Main model is model from

which all elements must be in Mirror model. For example, consistency validation checks if elements from one model e.g. (Use Case Actor) are represented in other model (e.g. Activity Swimline). Model consistency validation can be used to resolve for syntactic (e.g. misspelled element name) or semantics (missing element) error. There can be various Module elements are presented in Fig. 7 and the results are described in Table VI.
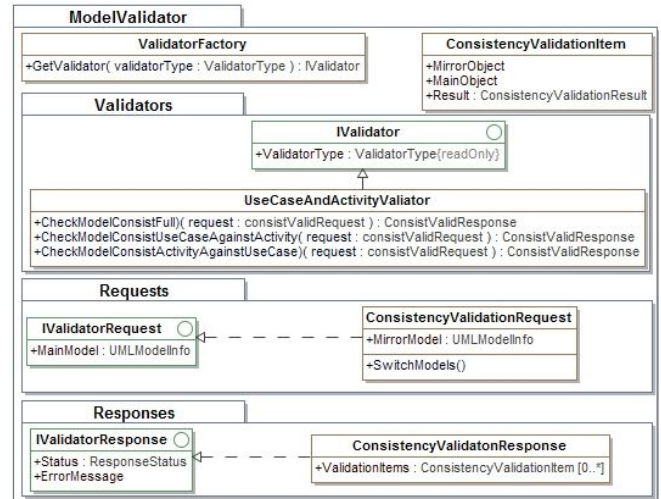

Fig. 7. ModelValidator module internal elements.

TABLE VI. MODEL VALIDATOR MODULE INTERNAL ELEMENTS.

Object	Description
IValidator	Interface for Validators (ValidatorType property).
UseCaseAndActivityValidator	Class that contains methods used for Validating UseCase and Activity models pair. Derived from validator class.
ConsistencyValidationRequest	Class used to provide validation input data to consistency validator
ConsistencyValidationResponse	Class used to return consistency validation results.
ConsistencyValidationItem	Class that contains validation result for single Main and Mirror objects

All newly introduced validators should implement IValidator interface and to be included into ValidatorFactory routines.

D. Mapper

This module provides UML models conversion to EM functionalities. Model *Mapper* takes as input *UMLModelObjectInfo* entities and creates *EnterpriseModel ObjectInfo* entities. A set of predefined and user selected options can be used for conversion. Model *Mapper* is created as static class as its functions are not model specific. Model *Mapper* has information about all *UMLModelObjectInfo* objects and *EnterpriseModelObjectInfo* objects and matches these objects using defined parameters. Module elements are presented in Fig. 8 and the results are described in Table VII.

TABLE VII. MODEL MAPPER MODULE INTERNAL ELEMENTS.

Object	Description
Mapper	Class that provides object mapping functionality. UMLModel entities are converted to Enterprise model entities using predefined and customer specified options.

| Object	Description
IMapperResponse	Interface for *Mapper* responses (Status and Error message properties).
IMapperRequest	Interface for *Mapper* requests.
MapperRequest	Class used to provide mapping input data to *Mapper*.
SingleObjectMapperResponse	Class used to return mapping result data for single object.
MapperResponse	Class used to return mapping result data.
SingleObjectMapperRequest	Class used to provide mapping input data to *Mapper*. Used for mapping single object.
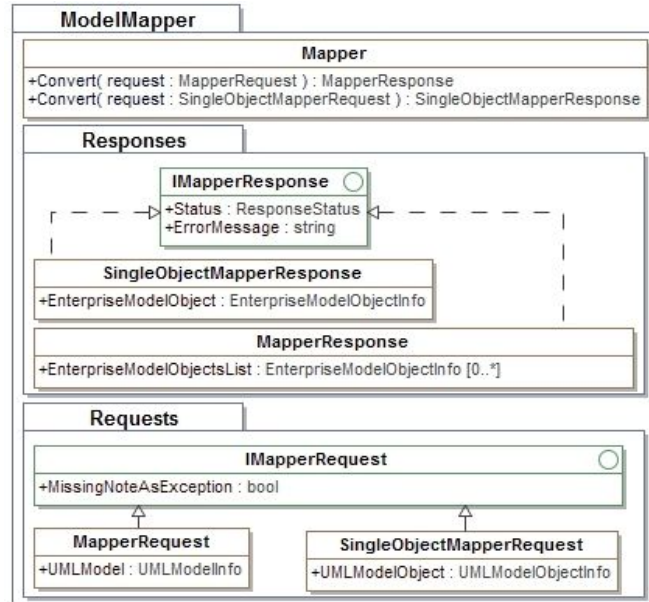
Fig. 8. ModelMapper module internal elements.
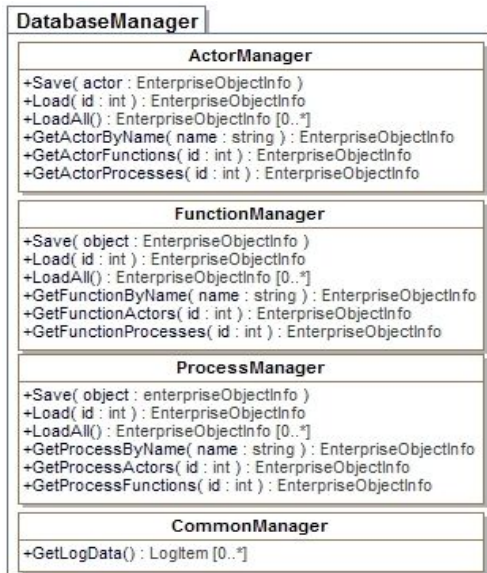
E. Database Manager

Fig. 9. DatabaseManager module internal elements.

DatabaseManager module provides database interaction functionalities. Each EM entity has its own data manager (e.g. *ActorManager, FunctionManager* etc.). Parts of the routines are common like save and load object by id, but most of them are dependent on object type. *CommonManager* is used for actions that are common for all objects or as helper for custom actions. Module elements are presented in Fig. 9 and the results are described in Table VIII.

TABLE VIII. DATABASEMANAGER MODULE INTERNAL ELEMENTS.

Object	Description
ActorManager	Class that contains *Actor* related DB functions and procedures. Provides data loading, saving, filtering, updating interface to DB.
FunctionManager	
ProcessManager	
CommonManager	Class that contains common or specific Database related functionalities. Basically is used as helper object.

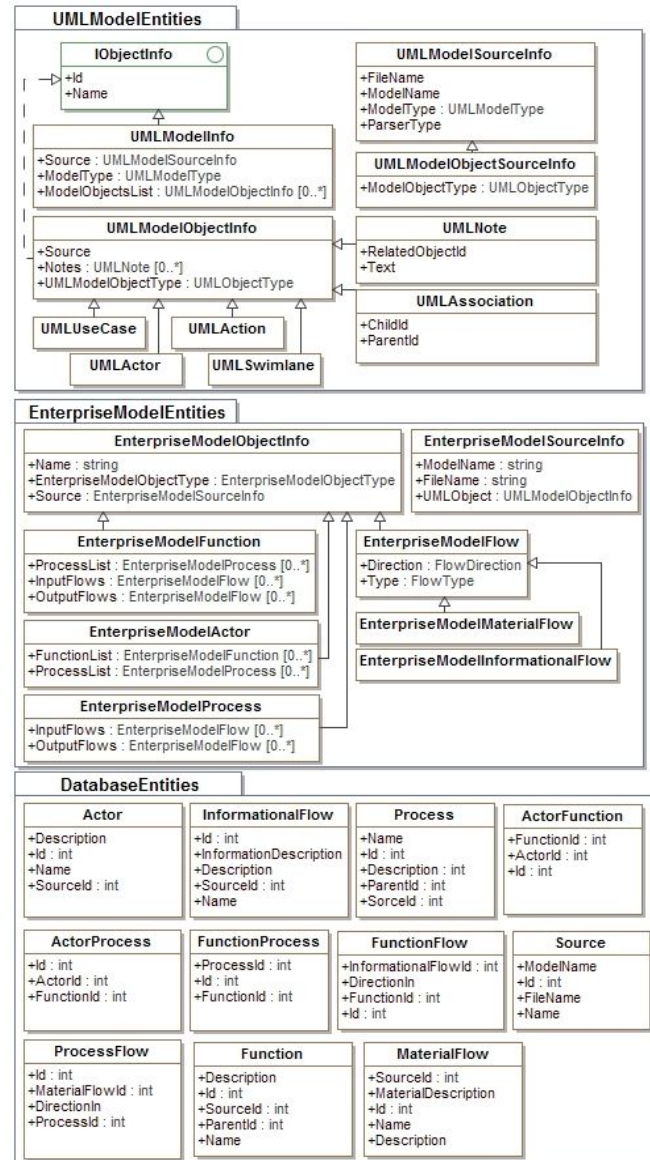*F. UML Model Entities, EM Entities, Database Entities and Constants*

Fig. 10. Internal elements for UMLModelEntities, EMEntities, DatabaseEntities modules.

In this chapter modules that contain data layer objects are described. *UMLModelEntities* module contains objects that are used for storing and manipulating UML model data. In fact, all parsed data are stored in these entities. Basically, the names of these entities are the same as UML model element type e.g. *UMLUseCase*, *UMLActor*, *UMLAction* etc. *UMLModelObjectInfo* is the base class from which all other classes are derived. This class contains basic UML model object's information such as Id, Name, *UMLObjectType* and source.

EnterpriseModelEntities module contains entities that are

created from *UMLModelEntities* after mapping process was performed. These entities have specific EM information that was appended during mapping process. Using EM Entities Enterprise Model validation and various analyses can be performed. There is one base element *EnterpriseModelObjectInfo* from which all other elements are derived. This element contains basic enterprise model element's information such as Name, Type and Source.

DatabaseEntities contains objects that are created by *ORM Mapper* (in particular case .Net Entity Framework). Each Database table has its own representing object. These objects provide functionality to manipulate database data in application. *DatabaseManager* classes have routines to query database tables, save, update or delete records as well as perform other actions. Module elements are presented in Fig. 10 and the results are described in Table IX.

TABLE IX. UML MODEL ENTITIES, EM ENTITIES, DATABASE ENTITIES INTERNAL ELEMENTS.

Object	Description
EnterpriseModelObjectInfo	Base class for all Enterprise Model objects. This class contains Object name, type and Source information.
EnterpriseModelSourceInfo	Class used to contain object origin information such as XMI file name, UML model name etc.
EnterpriseModelFunction, EnterpriseModelActor, EnterpriseModelFlow, EnterpriseModelProcess, EnterpriseModelMaterialFlow, EnterpriseModelInformationalFlow	Class that derives from *EnterpriseModelObjectInfo* class. Contains specific Function information.
IObjectInfo	Interface that contains Id and Name properties.
UMLModelObjectInfo	Base class for all UML model objects. This class implements *IObjectInfo* interface.
UMLModelInfo	Class that contains UML model information such as Model type, Model object list as well as model source.
UMLModelSourceInfo	Class that contains model source information such as File name, Model name, parser type and model object type.
UMLUseCase, UMlActor, UMLAction, UMLSwimline, UMLAssociation, UMLNote	Class that is derived from *UMLModelObjectInfo* class. Contains specific Note information.
Actor, Function, Process, FunctionProcess, ActorFunction, ActorProcess	Auto generated classes by ORM mapper. Used as Database record representation in application. These classes have all database fields including *key* fields and *foreign key* fields.

The described entities provide architecture backbone for prototype to be able to read Use Case and Activity models information from XMI file, validate these models, map to enterprise model elements as well as perform Database related actions.

Constants module is created as main source of static information (such as model types definitions, object types definitions, messages, error codes etc.). Data are saved as Constant or Enum variables. In most cases this module is reused by *Logic Layer's* objects.

VIII. CONCLUSIONS

In the first chapters of article the detailed SysML models (Use Case, Block Definition, Activity, Requirements)

parsing algorithms that are necessary for KB-MDA IS engineering method are presented. These algorithms use recursive data acquisition method. The defined solution ensures data consistency among particular Use Case, Activity, Block Definition, Requirements models thus providing more accurate user requirements specification process.

The next chapters deals with implementation of these algorithms by The KB-MDA tool's prototype. The prototype described in the article is capable of parsing, validating, mapping Use Case and Activity models to Enterprise model. The future works are: to include more UML models (e.g. Class) to this process and to define more validation options as well as introduce advanced enterprise model analysis features.

REFERENCES

[1] OMG UML, *Unified Modeling Language version 2.5. Unified Modeling,* 2012. [Online]. Available: http://www.omg.org/spec/UML/2.5/Beta1
[2] OMG MOF, *Meta-Object Facility*, 2013. [Online]. Available: http://www.omg.org/spec/MOF
[3] M. S. Soares, J. Vrancken, "Model-driven user requirements specification using SysML", *Journal of Software*, vol. 3, no. 6, pp. 57–68, 2008. [Online]. Available: http://dx.doi.org/10.4304/jsw.3.6.57-68
[4] A. Lopata, M. Ambraziunas, S. Gudas, R. Butleris, "The main principles of knowledge-based information systems engineering", *Elektronika ir elektrotechnika*, vol. 11, pp. 99–102, 2012.
[5] A. Morkevicius, S. Gudas, "Enterprise knowledge based software requirements elicitation", *Information Technology and Control*, vol. 40, no. 3, pp. 181–190, 2011. [Online]. Available: http://dx.doi.org/10.5755/j01.itc.40.3.626
[6] A. Lopata, M. Ambraziunas, S. Gudas, "Knowledge based MDA requirements specification and validation technique", *Transformations in Business & Economics,* vol. 11, no. 1(25), pp. 248–261, 2012.
[7] M. Kirikova, A. Finke, J. Grundspeni, "What is CIM: an information system perspective", *Advances in Databases and Information Systems,* vol. 5968, pp. 169–176, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12082-4_22
[8] A. Fouad, K. Phalp, J. M. Kanyaru, S. Jeary, "Embedding requirements within Model-Driven Architecture", *Software Quality Journal*, vol. 19, no. 2, pp. 411–430, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11219-010-9122-7
[9] A. Rodriguez, E. Fernandez-Medina, M. Piattini, "CIM to PIM Transformation: A Reality", *Research and Practical Issues of Enterprise Information Systems II IFIP International Federation for Information Processing,* vol. 255, pp. 1239–1249, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-76312-5_50
[10] M. Kardos, M. Drozdova, "Analytical method of CIM to PIM transformation in Model Driven Architecture (MDA)", *Journal of Information and Organizational Sciences*, vol. 34, pp. 89–99, 2010.
[11] Wei Zhang, Hong Mei, Haiyan Zhao, Jie Yang, "Transformation from CIM to PIM: A feature-oriented component-based approach", *Model Driven Engineering Languages and Systems – Series: Lecture Notes in Computer Science*, vol. 3713, pp. 248–263, 2005.
[12] J. Dunkel, R. Bruns, "Model-Driven architecture for mobile applications", in *Proc. 10th Inter-national Conf. Business Information Systems (BIS)*, vol. 4439, pp. 464–477, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72035-5_36
[13] R. Vitiutinas, D. Silingas, L. Telksnys, "Model-driven plug-in development for UML based modelling systems", *Information Technology and Control*, vol. 40, no. 3, pp. 191–201, 2011. [Online]. Available: http://dx.doi.org/10.5755/j01.itc.40.3.627
[14] No Magic, Inc., *Unified Modeling Language (UML), BPMN, SysML, UPDM, SOA, Business Process Modeling Tools.* 2013. [Online]. Available: http://www.nomagic.com
[15] OMG XMI MOF 2 XMI Mapping, 2013. [Online]. Available: http://www.omg.org/spec/XMI
[16] T. Ndie, C. Tangha, F. Ekwoge, "MDA (Model-Driven Architecture) as a software industrialization pattern: an approach for a pragmatic software factories", *Journal of Software Engineering and Applications*, vol. 3, no. 6, pp. 561–571, 2010. [Online]. Available: http://dx.doi.org/10.4236/jsea.2010.36065