DOI:10.2298/CSIS120203034C

A MOF based Meta-Model and a Concrete DSL Syntax of IIS*Case PIM Concepts

Milan Čeliković, Ivan Luković, Slavica Aleksić, and Vladimir Ivančević

University of Novi Sad, Faculty of Technical Sciences,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia
{milancel, ivan, slavica, dragoman}@uns.ac.rs

Abstract. In this paper, we present a platform independent model (PIM) of IIS*Case tool for information system (IS) design. IIS*Case is a model driven software tool that provides generation of executable application prototypes. The concepts are described by Meta Object Facility (MOF) specification, one of the commonly used approaches for describing meta-models. One of the main reasons for having IIS*Case PIM concepts specified through the meta-model, is to provide software documentation in a formal way, as well as a domain analysis purposed at creation a domain specific language to support IS design. Using the PIM meta-model, we can generate test cases that may assist in software tool verification**.** The meta-model may be also a good base for the process of the concrete syntax generation for some domain specific language.

Keywords: information system modeling, domain specific languages, domain specific modelling, platform independent models.

1. Introduction

IIS*Case is a software tool that provides a model driven approach to information system (IS) design. It supports conceptual modeling of database schemas and business applications. IIS*Case, as a software tool assisting in IS design and generating executable application prototypes, currently provides:

- Conceptual modeling of database schemas, transaction programs, and business applications of an IS;
- Automated design of relational database subschemas in the 3rd normal form (3NF);
- Automated integration of subschemas into a unified database schema in the 3NF;
- Automated generation of SQL/DDL code for various database management systems (DBMSs);
- Conceptual design of common user-interface (UI) models; and
- Automated generation of executable prototypes of business applications.

In order to provide design of various platform independent models (PIM) by IIS*Case, we created a number of modeling, meta-level concepts and formal rules that are used in the design process. Besides, we have also developed and embedded into IIS*Case visual and repository based tools that apply such concepts and rules. They assist designers in creating formally valid models and their storing as repository definitions in a guided way. Main features of IIS*Case and the specification of its usage may be found in [1].

There is a strong need to have PIM concepts specified formally in a platform independent way, i.e. to be fully independent of repository based specifications that typically may include some implementation details. Our current research is based on two related approaches to formally describe IIS*Case PIM Concepts. One of them is based on Meta Object Facility (MOF) and the other one on a textual Domain Specific Language (DSL). In [2], we give a specification of the IIS*Case textual modeling language, named IIS*CDesLang that formalizes IIS*Case PIM concepts and provides modeling in a formal way. IIS*CDesLang meta-model is developed under a visual programming environment for attribute grammar specifications named VisualLISA [3].

In [4] we propose a meta-model of IIS*Case PIM concepts, which is based on the Meta Object Facility (MOF) 2.0. MOF 2.0 is a common meta-meta-model proposed by Object Management Group (OMG) where meta-models are created by the use of UML class diagrams and Object Constraint Language (OCL) [5]. As we could not find standardized implementation of MOF, we decided to use Ecore meta-meta-model. Ecore is the Eclipse implementation of MOF 2.0 in Java programming language which is provided by Eclipse Modeling Framework (EMF) [6]. Ecore concepts are not always identical to MOF 2.0 concepts, but they are expressive enough to create our IIS*Case meta-model. A benefit of such a meta-model is providing software documentation in a formal way. Besides, created meta-model can be used for the software tool verification in EMF environment. It also represents a domain analysis specification necessary to create IIS*CDesLang, as a textual DSL that supports IS design. In this paper we give an example that illustrates the process of modeling a part of an IS using IIS*Case PIM concepts. We also present a small part of a concrete syntax grammar that is based on the definition of IIS*Case PIM concepts.

In Fig. 1 we illustrate the four layered architecture of our solution, which is tailored from OMG four–layered architecture standard. Level M3 comprises meta-meta-model (MOF 2.0) [7] that is used for implementation of the IIS*Case meta–model (M2). M2 level represents the IIS*Case PIM meta-model specified by MOF specification and implemented in EMF. Using the IIS*Case PIM meta-model, a designer specify and implement a conceptual model of an IS that is placed at the M1 level of the four-layered data architecture from Fig. 1. By using IS applications generated by IIS*Case, end-users manipulate real data, i.e. they create and use models of entities from real world (M0), using the conceptual model (M1).

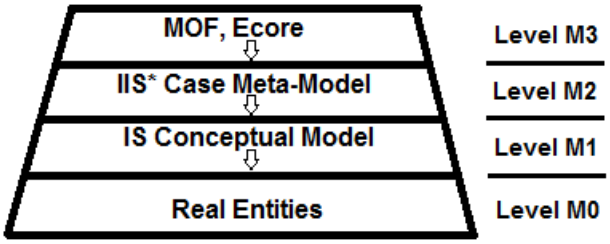

Fig. 1. Four layered meta-data architecture

Apart from Introduction and Conclusion, the paper is organized in four sections. In Section 2, we present a related work. In Section 3 we give a presentation of IIS*Case PIM concepts specified through the meta-model that is implemented in EMF environment. In Section 4 we illustrate an example of IIS*Case PIM concepts usage, while in Section 5 we give a concrete syntax definition of main PIM concepts.

2. Related Work

Nowadays, meta-modeling is widely spread area of research and there is a huge number of references covering MOF based meta-models. However, we could not find papers presenting formal approaches to the specification of meta-model implementation and design of CASE tools, based on MOF or Ecore meta-meta-models.

We found a vast number of meta-model specifications and implementations based on MOF or Ecore specifications. Meta-models based on MOF are also presented in [8] and [9]. The authors in both papers propose the meta-models of the Web Modeling Language. The meta-model specification and design is implemented under EMF environment. Defining W2000 [8] as a MOF meta-model, the authors specify it as an UML profile. In [9], the authors provide a solution for the generation of MOF meta-models from document type definition (DTD) specifications [10]. A formal specification of OCL is given in [11]. In their meta-model, the authors precisely define the syntax of OCL, as it is given in [5]. They propose a solution for the presented meta-model integration with the UML meta-model. In [12], the authors propose the Kernel MetaMetaModel (KM3) representing a DSL for meta-model definition. In [13], the authors propose the UML Profile, EUIS, used for the specification of business applications' user interfaces. Their solution provides automatic interface code generation that is based on their own HCI standard. They developed a DSL specified as UML Profile that offers user interface modeling and generation. In [14] the authors propose a solution for the kiosk applications development. They present KAG, a DSL that provides kiosk applications development in a more rapid way than standard high-level programming languages. While the presented DSL provides rapid application

prototyping of new applications, it also simplifies the maintenance process of existing applications. The DSL has also reduced the number of errors that were common in the process of programming using standard high level programming languages. The authors of the paper [15] present the DOMMLite, a DSL that provides the definition of database applications' static structures. The language structure has been defined at the level of the meta-model. The textual syntax has been defined in order to provide creation, update and persistence of DOMMLite models. They have also developed a textual Eclipse editor that provides generation of source code for graphical-user interface forms supporting CRUDS (Create-Read-Update-Delete-Search) operations. In [16], the authors present a selection of 75 key publications, covering the area of DSLs. They give an overview of the terminology, DSL examples, design methodologies, and implementation techniques. In [17], the authors give an overview of the problems in the decision, analysis, design, implementation and deployment phases of DSL development. They have identified patterns for the first four phases that can aid DSL developers. They have also presented language development systems and frameworks aimed at facilitating the development process. The authors of the paper [18] present Sequencer, a domain specific modeling language for programming or modeling measurement procedures without interacting with programming engineers. Sequencer provides development of measurement procedures inside the measurement system DEWESoft using DCOM objects. It is a DSL that provides textual or visual mode, customized for the application development in the measurement domain. Similar to the papers discussed in this section, we base our research on the development of the DSL in the domain of IS development. In this paper we focus on the meta-model specification of IIS*Case PIM concepts and the generation of concrete syntax.

There are various meta-modeling tools that are generally based on their own meta-meta-model specifications. One of them is Generic Modeling Environment (GME) [19], a configurable toolkit for domain specific modeling and program synthesis based on UML meta-models. MetaEdit+ [20], [21] and [22] allows creation and design of meta-models by the use of a graphical editor providing the Graph-Object-Property-Port-Role-Relationship data model. All of these tools may also be used for the IIS*Case PIM meta-model description in a formal way.

3. IIS*Case Meta-Model

IIS*Case provides a definition of several concepts embedded into IIS*Case repository, that typically may include some implementation details. In this paper, we present only IIS*Case PIM meta-model concepts specified by Ecore meta-meta-model. Hereby we overview here the following main IIS*Case PIM concepts: *Project, Domain* and *Attribute* as *Fundamental concepts*, *Program unit, Application system*, *Application type, Form type* and *Component type*. A model of the IIS*Case main concepts with their properties

and relationships is presented latter on, in Fig. 2. More information about these concepts may be found in [1] and [23], as well as in many other authors' references.

3.1. Project

In IIS*Case, modeling process is organized through one or more projects. Therefore, the central concept in our meta-model from Fig. 2 is *Project*. For each project, a designer defines the project name as its mandatory property. All existing elements in the repository of IIS*Case are always created in the context of a project. *Fundamental concepts* and *Application systems* are subunits of a *Project*. *Fundamental concepts* are formally independent of any application system. *Fundamental concept* instances can be used in more than one application system, because they are defined at the level of a project. *Fundamental concepts* comprise zero or more:

- *Attributes*,
- *Domains*,
- *Program units* and
- *Inclusion dependencies*.

Each project is organized through application systems and fundamental concepts. For each project, we can define zero, or more instances of the *Application system* concept. An IS designer may create application systems of various types. By the *Application type* concept, a designer may introduce various application system types and then associate each instance of an application system to exactly one application type.

At the level of a project, IIS*Case provides generation of various types of repository reports. As the Report is not a real modeling concept, it does not belong the IIS*Case PIM concepts. However, the IIS*Case repository contains Report concept. It is used by the IIS*Case reporting tools.

3.2. Domain

Domains specify allowed values of database attributes. They are classified as:

- Primitive and
- User defined.

Therefore, in our meta-model, there are two classes: *PrimitiveDomain* and *UserDefinedDomain* that are subclasses of a *Domain* class.

Primitive domains represent primitive data types that exist in formal languages, such as string, integer, char, etc. The reason behind the existence of user defined domain concept is to allow designers to create their own data types in order to raise the expressivity of their models. Each domain has its name, description and default value. At the level of a primitive domain, a designer may specify *length required* item value. It denotes if a numeric length: must be, may be, or is not to be given. For user defined domains, a

designer needs to define a domain type and a check condition. IIS*Case supports two classes of user defined domains:

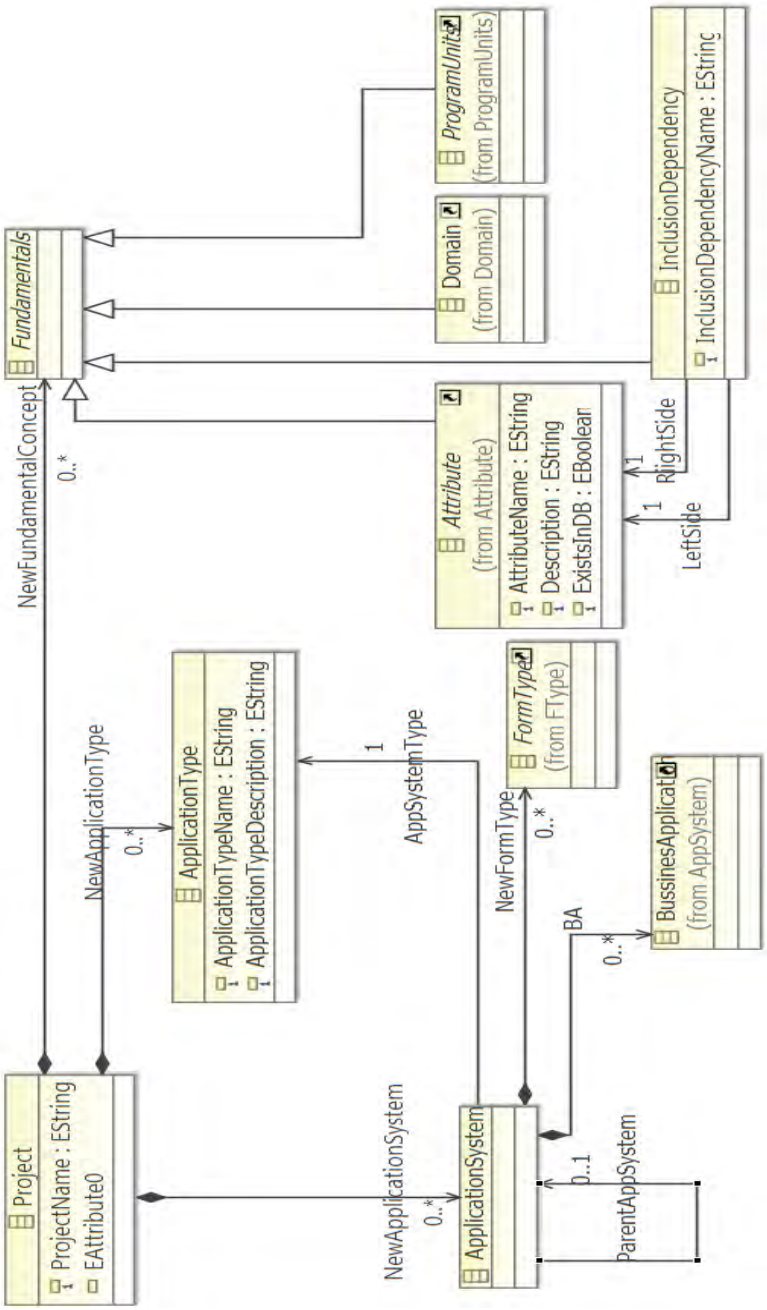

Fig. 2. A meta-model of IIS*Case main PIM concepts

- Domains created by the inheritance rule and
- Complex domains.

A domain created by the inheritance rule references a specification of some primitive or user defined domain. We call it a child domain, while the referenced domain is also called a superordinated or parent domain. By using the inheritance, all the rules defined at the level of a parent domain also hold for the child domain.

Complex domains may be created by the tuple rule, set rule, or choice rule. A domain created by the tuple rule we simply call the tuple domain, because it represents a tuple of values. The items of such a tuple structure are some of already created attributes. A domain created by the choice rule we call a choice domain. It is specified in almost the same way as a tuple domain. The choice domain concept is the same as the choice type of XML Schema Language. Each value of a choice domain corresponds to exactly one attribute. A set domain represents sets of allowed values over a specified domain.

Check condition is a regular expression that additionally constrains possible values of a domain created by a designer.

Domain concept allows definition of display properties of screen items that correspond to attributes and their domains. Each domain corresponds to exactly one element of the *Display* type. The *Display* concept specifies rules, later used by the application generator to generate screen or report items. Generated screen or report items correspond to some of the attributes, and attributes correspond to some of domains. Technical aspects of the display properties implementation may be found in [24] and [25].

3.3. Attribute

In Fig. 3, we present a meta-model of the *Attribute* concept. Each attribute in a project is identified by its name. It also has a description and a Boolean property specifying if it belongs to the database schema. In practice, the most of created attributes belong to the database schema. For attributes representing derived (calculated) values in reports or screen forms a designer may decide if they are to be included in the database schema. By this, we classify attributes as: a) included or b) non-included in a database schema.

According to the way how an attribute gains a value, we classify attributes as: a) non-derived or b) derived. A value of a non-derived attribute is created by an end-user. A value of derived attribute is always calculated from the values of other attributes, by applying some function, i.e. a calculation formula. There is a rule that any non-included attribute must be specified as derived one.

A function that is used to calculate a derived attribute value is formally specified in the IIS*Case repository. Additionally, a designer may specify parameters that are passed to the function. The *Function* concept will be presented in the next subsection, Program Units. If an attribute is non-included in a database schema, the function is referenced as a query function.

Only derived attributes that are included in a database schema may additionally reference three IIS*Case repository functions specifying how to calculate the attribute values on the following database operations: insert, update and delete.

An attribute may be specified as a) elementary or b) renamed. A renamed attribute references a previously defined attribute. The source of such an attribute is the referenced attribute, but with the different semantics. The renamed attribute needs to be included in database schema. Renaming is a concept that also exists in the Entity-Relationship and relational data models. By means of renaming, a designer may differentiate between semantics of "similar" attributes. If a designer introduces a new attribute A1 and specifies it as a renamed from the existing attribute A, he or she actually specifies an inclusion dependency of the form [A1] $\subseteq$ [A] at the level of a universal relation scheme. More information about the use of renaming concept in the context of IIS*Case tool may be found in [1]. Inclusion dependency is modeled in Fig. 2 in our meta-model as the class *InclusionDependency* inheriting *Fundamentals.* It is also related with class Attribute over two relationships, that actually represent left and right side of the inclusion dependency.

To each attribute a domain must be associated. This association allows defining a default value and a check condition. If the attribute value is not specified, the default value is assigned to it. Check condition is the attribute check expression that represents the regular expression that additionally constrains the value of the attribute.

At the level of an attribute, we can specify the display properties. The concept of the *Display* properties is same as the one at the level of the *Domain* concept. Values of display properties, specified at the level of the associated domain, may be inherited or overridden according to the requirements of an IS project.

3.4. Program Units

The *Program unit* concept is used to express complex application functionalities. We classify program units as: a) *Functions*, b) *Packages* and c) *Events*.

The *Function* concept is used to specify any complex functionality that later may be used in other specifications. Each function has its name and return type that are mandatory properties, as well as a formal specification of a function body and a description that are optional. The return type is a reference to a domain. A function specification may include a list of formal parameters. Each formal parameter of a function is specified by its name and a sequence number, as mandatory properties. Exactly one domain is associated with each formal parameter. Any parameter may also have a default value specified. With respect to the ways of exchanging values between the function and its calling environment, we classify formal parameters as: a) In, b) Out and c) In-Out, with a usual meaning as it is in many general purpose programming languages.

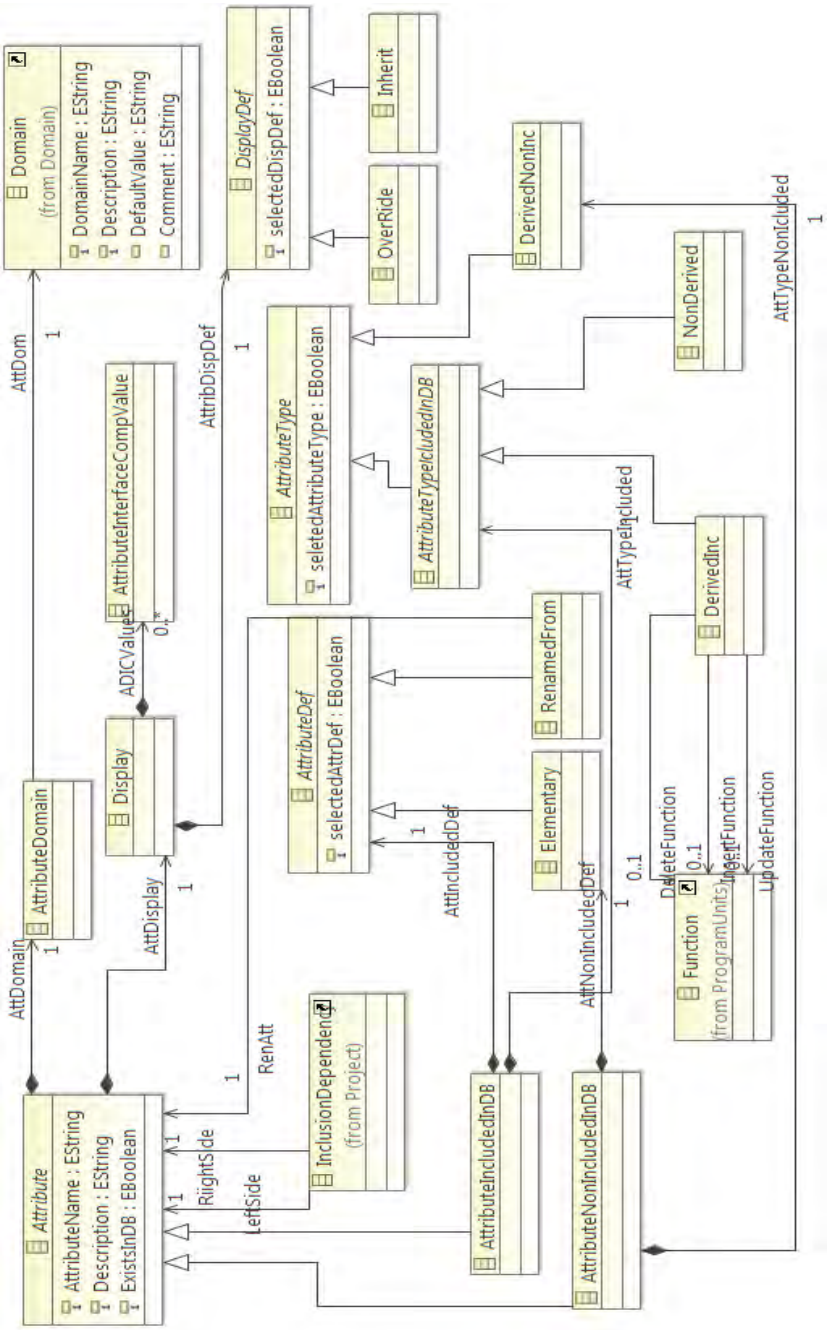

Fig. 3. A meta-model of the IIS*Case Attribute concept

IIS*Case provides grouping of created functions into packages. Each function may be included into one or more packages, or may stay as a stand-alone object. By the location of the deployment in a multi-layer architecture, the packages are classified as: a) Database server packages, b) Application server packages and c) Client packages. A package is identified by its name, and may have an optional description.

The *Package* concept is modeled by the inheritance rule. We have the abstract class named *Package*. It is superordinated to the classes: *DBServerPackage*, *ApplicationServerPackage* and *ClientPackage*. For each instance of the *Package* class, there may be zero or more references to the instances of the *Function* class.

The *Event* concept is used to represent any software event that may trigger some action under a specified condition. Each event is identified by its name, and may have an optional description. Similar to the packages, by the location of the deployment in a multi-layer architecture, we also classify events as: a) Database server events, b) Application server events and c) Client events. The *Event* concept is modeled in the similar way like *Package*, by applying the inheritance rule.

3.5. Application System

The *Application System* concept is used to model organizational parts of each Project. Each application system has its name and a description as mandatory properties. Besides, it may reference other, subordinated application systems that we call child application systems. By this, a designer may create a hierarchy of application systems in a project. Application system hierarchy is modeled by a recursive reference.

Various kinds of IIS*Case repository objects may be created at the level of an application system, but in this paper we focus on two of them only, as PIM concepts: a) *Form type* and b) *Business Application*.

3.6. Form type

Form type is the main concept in IIS*Case. The meta-model of this concept is presented in Fig. 4. It abstracts document types, screen forms, or reports that end-users of an information system may use in a daily job. By means of the *Form type* concept, designers indirectly specify at the level of PIMs a model of a database schema with attributes and constraints included. At the same time, they also specify a model of transaction programs and applications of an information system.

Apart from creating form types in application systems, designers may include into their application systems form types created in other application systems being modeled. Therefore, we classify form types as: a) owned and b) referenced. A form type is owned if it is created in an application system. It

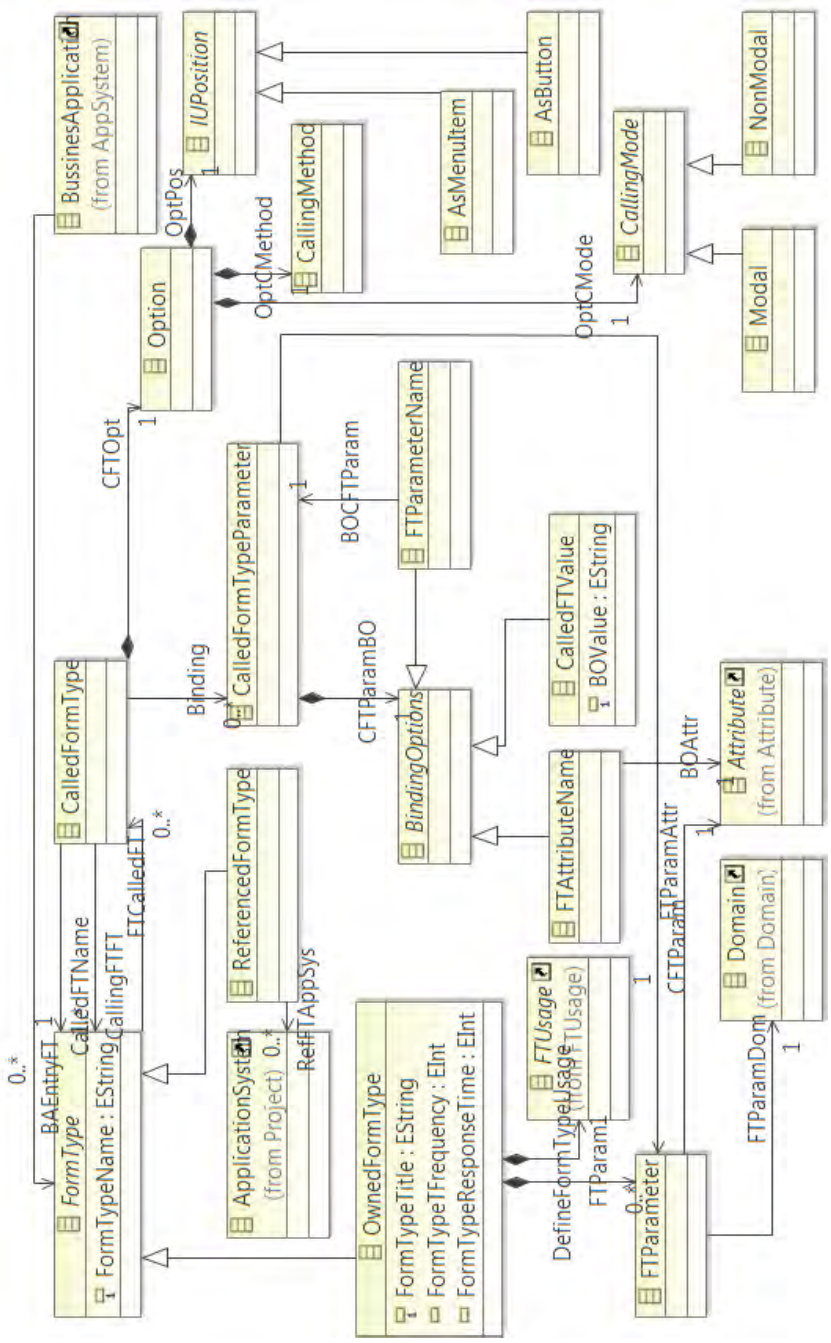

Fig. 4. A meta-model of the IIS*Case Form Type concept

may be modified later on through the same application system without any restrictions. A referenced form type is created in another application system and then included into the application system being considered. All the referenced form types in an application system are read-only.

Each form type has a name that identifies it in the scope of a project, a title, frequency of usage, response time and usage type. Frequency is an optional property that represents the number of executions of a corresponding transaction program per time unit. Response time is also an optional property specifying expected response time of a program execution. By the usage type property, we classify form types as: a) menus and b) programs.

Menu form types are used to model menus without data items. Program form types model transaction programs providing data operations over a database. They may represent either screen forms for data retrievals and updates, or just reports for data retrievals. As a rule, a user interface of such programs is rather complex. A program form type may be designated as *considered in database schema design* or *not considered in database schema design*. Form types considered in database schema design are used later as the input into the database schema generation process. Form types not considered in database schema design are not used in the database schema generation process. They may represent reports for data retrievals only.Each program form type is a tree of component types. A component type has a name, title, number of occurrences, allowed operations and a reference to the parent component type, if it is not a root component type. Name is the component type identifier. All the subordinated component types of the same parent must have different names.

Each instance of the superordinated component type in a tree may have more than one related instance of the corresponding subordinated component type. The number of occurrences constrains the allowed minimal number of instances of a subordinated component type related to the same instance of a superordinated component type in the tree. It may have one of two values: 0-N or 1-N. The 0-N value means that an instance of a superordinated component type may exist while not having any related instance of the corresponding subordinated component type. The 1-N value means that each instance of a superordinated component type must have at least one related instance of the subordinated component type.

The allowed operations of a component type denote database operations that can be performed on instances of the component type. They are selected from the set {*read*, *insert*, *update*, *delete*}.

A designer can also define component type display properties that are used by the program generator. The concept of component type display is defined by properties: window layout, data layout, relative order, layout relative position, window relative position, search functionality, massive delete functionality and retain last inserted record.

Window layout has two possible values: “New window” and “Same window” and specifies if the component type is to be placed in a new window or in the same window as the parent component type. Data layout specifies the way of component type representation in a screen form. Two values are possible:

"Field layout" or "Table layout". By the "Field layout", only one record at a time is displayed in a form. By the "Table layout", a set of records at a time is displayed in a screen form, in a form of a table. The relative order is a sequence number representing the order of a component type relative to the other sibling component types of the same parent in a form type tree. The layout relative position represents the component type relative position to the parent component type. We may select "Bottom to parent" value if we want to place the component type below the layout of the parent component type in a generated screen form, or "Right to parent" value if it is to be placed right to the parent one. Window relative position is to be specified only when "New window" layout is selected. A designer may specify one of the three possible values: "Center", "Left on top", or "Custom". The "Center" value denotes that the center of a new window is positioned to match the center of the parent window. "Left on top" specifies that the top left corner of the new window will match the top left corner of the parent window. By selecting the "Custom" value, a relative position of the new window top left corner to the top left corner of the parent window is explicitly specified by giving X and Y relative positions.

"Search functionality" represents the Boolean property that enables generation of the filter for data selection. If search functionality is enabled, end-users are allowed to refine the WHERE clause of a SQL SELECT statement. If checked, "massive delete functionality" provides a generation of a delete option next to each record in a table layout. The "retain last inserted record" property specifies if the last inserted record is to be retained on the screen for future use.

Each component type includes one or more attributes. A component type attribute is a reference to a project attribute from the Fundamentals category. It has a title that will appear in the generated screen form. Also, it may be declared as mandatory or optional on the screen form. The allowed operations of a component type attribute denote database operations that can be performed on the attribute, by means of the corresponding screen item. They are selected from the set {*query*, *insert*, *update*, *nullify*}. For a component type attribute a designer may also specify display properties and by this define its presentation details in the screen form. The display properties are specified in the same way as it is for attribute specifications. Values of the display properties may be inherited from the attribute specification or overridden.

So as to unify the layout formatting rules of selected component type attributes, a designer may group them into items groups. Each item group may include one or more component type attributes or other item groups from the same component type. Any item group has its name, title, context and overflow properties. The name and title are mandatory properties. Context and overflow are Boolean properties, specifying if an item group is to be used for presenting layout contextual information or as a layout overflow area.

Each component type attribute provides defining a "List of values" (LOV) functionality. To do that, a designer needs to reference a form type that will serve as a LOV form type. He or she should also define how an end-user can edit attributes: "Only via LOV" or "Directly & via LOV". "Only via LOV" property

means that attribute value may not be inserted or edited using a keyboard, but only using the LOV. “Directly & via LOV” means that inserting or editing attribute values is provided both via keyboard and LOV. “Filter value by LOV” property specifies if all values from LOV will be displayed, or only those filtered according to the pattern given by an end-user. Restrict expression represents the where clause that is concatenated to the rest of where clause in the SQL statement supporting the LOV.

Each component type has one or more keys. Each component type key comprises one or more component type attributes. It represents the unique identification of a component type instance but only in the scope of its superordinated component instance. Uniqueness constraints may be defined for each component type also. Each component type uniqueness constraint comprises at least one component type attribute, but may have more than one. If uniqueness constraint attributes have non-null values, it is possible to uniquely identify a component type instance but only in the scope of the superordinated component instance.

3.7. Business Application

Business Application concept represents the way to formally describe an IS functionality and is organized through a structure of form types. Each business application has a name and a description. One of the form types included into the structure must be declared as the entry form type of the application. It represents the first transaction program invoked upon the launching of the application. Each business application must have the entry form type. To create the form type structure of an application, a concept of the form type call is used. By the form type calls, designers model execution of calls between generated transaction programs. They are also used to model parameters and passing the values between two transaction programs during the call executions. The concept of a form type call comprises two form types: a calling form type and a called form type.

Any form type may have formal parameters defined. Each formal parameter has a mandatory name as the identifier. It must be related to exactly one domain. In the specification of a form type call, it is possible to associate each parameter to a called form type attribute. By this, a designer specifies to which attributes real parameter values will be passed during the call execution.

For a called form type in a call we need to specify Binding and Options properties. Binding property comprises formal parameters of a called form type. For each parameter a designer specifies how a real argument value is to be passed to the parameter. There are three possible options: “value”, “attribute reference”, or “parameter reference”. The value is a constant that will be passed during a call execution. The “attribute reference” provides a relation to a calling form type attribute that gives a value to be passed to the parameter during a call execution. The “parameter reference” provides a

relation to a calling form type parameter that gives a value to be passed to the parameter during a call execution.

The Options properties comprise: calling method, calling mode, and UI position. Calling method comprises two Boolean properties: a) “Select on open” and b) ”Restricted select”. “Select on open” means that the called form type is opened with an automatic data selection. “Restricted select” allows the data selection in the called form type restricted just to the values of passed parameters. Calling mode specifies a general behavior of the calling form type during the call execution. Three possibilities are allowed: “Modal”, “Non-modal” or “Close calling form”. “Modal” means that a user cannot activate the calling form type while the called form type is opened. “Non-modal” means that both the calling and the called form type are simultaneously active in the screen. “Close calling form” is used to cause the closing of the calling form type during the call execution. UI position specifies how a call will be provided at the level of UI: as a menu item or as a button item.

4. IIS*Case PIM Concepts Usage

For many years, IIS*Case provides visually oriented tools for the IS specification in a formal way. In this section we present a different approach where an IS is modeled using the IIS*Case PIM concepts specified at the level of meta-model in EMF. EMF is not only the framework that provides modeling at the level of meta-models, but also supports model implementations based on the created meta-models. In this section, by an example we illustrate the usage of some PIM concepts belonging to our meta-model trough EMF.

In Fig. 5 we present a part of the project Student Service IS. It represents a form type Student_Grades that refers to information about students’ grades. In the following text the project and its main parts are explained in more details.

Using the Eclipse Modeling Framework (EMF), end-users are able to specify the model of Student Service IS using the IIS*Case PIM concepts. In Fig. 6, we present a part of the formal specification of Student Service IS in a form of a tree structure, created by means of the PIM concepts modeled in EMF. It represents the form type from Fig. 5. In the following text we also explain the model from Fig. 6 in more details.

Modeled IS consists of two application systems: *Student Service* and *Faculty Organization*. *Student Service* application system, referenced in Fig. 5 in the upper left rectangle, is a child application system of the parent application system *Faculty Organization* that is referenced in the upper right corner of Fig. 5. In Fig. 6 we have defined the Project, where a value of the *Name* property is *FacultyIS*. We have also defined at the level of the Project two kinds of application types: a) *System* and b) *Subsystem*. Further, we classified application system *Faculty Organization* as the *System* and *Student Service* as *the Subsystem* application type. In Fig. 6, at the level of the Project

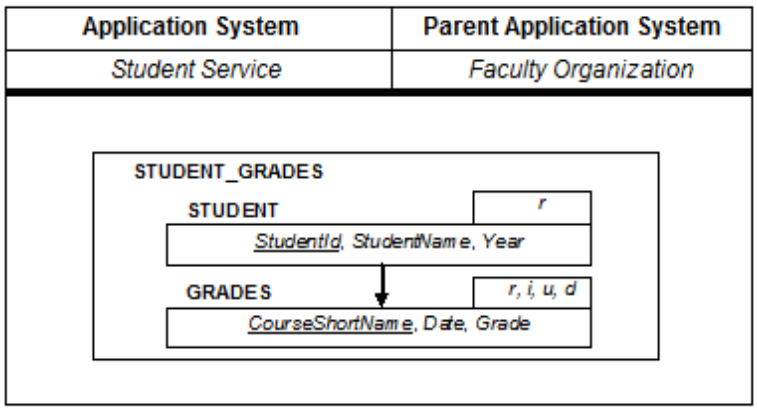

Fig. 5. Application system Student Service

FacultyIS, we have also created a set of attributes, including: *StudentID*, *StudentName*, *Year*, *CourseShortName*, *Date* and *Grade*. The set of these attributes is defined in the Fundamentals category. The attributes defined in the Fundamentals category are later used in the specification of other IS components.

Further, we illustrate the usage of the Form Type concept. We have the form type *Student_Grades,* placed inside the main area of Fig. 5. It has two component types: *Students* and *Grades*. *Student_Grades* form type is presented in Fig. 6 as the Owned Form Type *STG – Student Grades* at the level of the application System *Student Service.* It refers to the information about student grades.

The rectangles that represent *Student* and *Grades* component types are located inside the rectangle representing the form type *Student_grades*. While *Student* component type represents instances of students, *Grades* component type represents instances of grades for each student. *Student* component type is the parent to the *Grades* component type. *Student* and *Grades* component types are modeled in Fig. 6 at the level of the Owned Form Type *STG – Student Grades.*

Allowed database operations for the component type are: read, update, insert and delete. They are presented in Fig. 5 with the abbreviations: r, u, i, d, respectively. The only allowed database operation for *Student* is read, while the allowed operations for *Grades* are read, insert, update and delete. The allowed database operations for the component types are specified in our Project modeled in EMF, although they could not be seen in Fig. 6. End-users of the generated transaction program specified by the form type *Student_Grades* will be able to read data about student instances. They may read, update and delete existing grades for each student, as well as insert new instances of the grades.

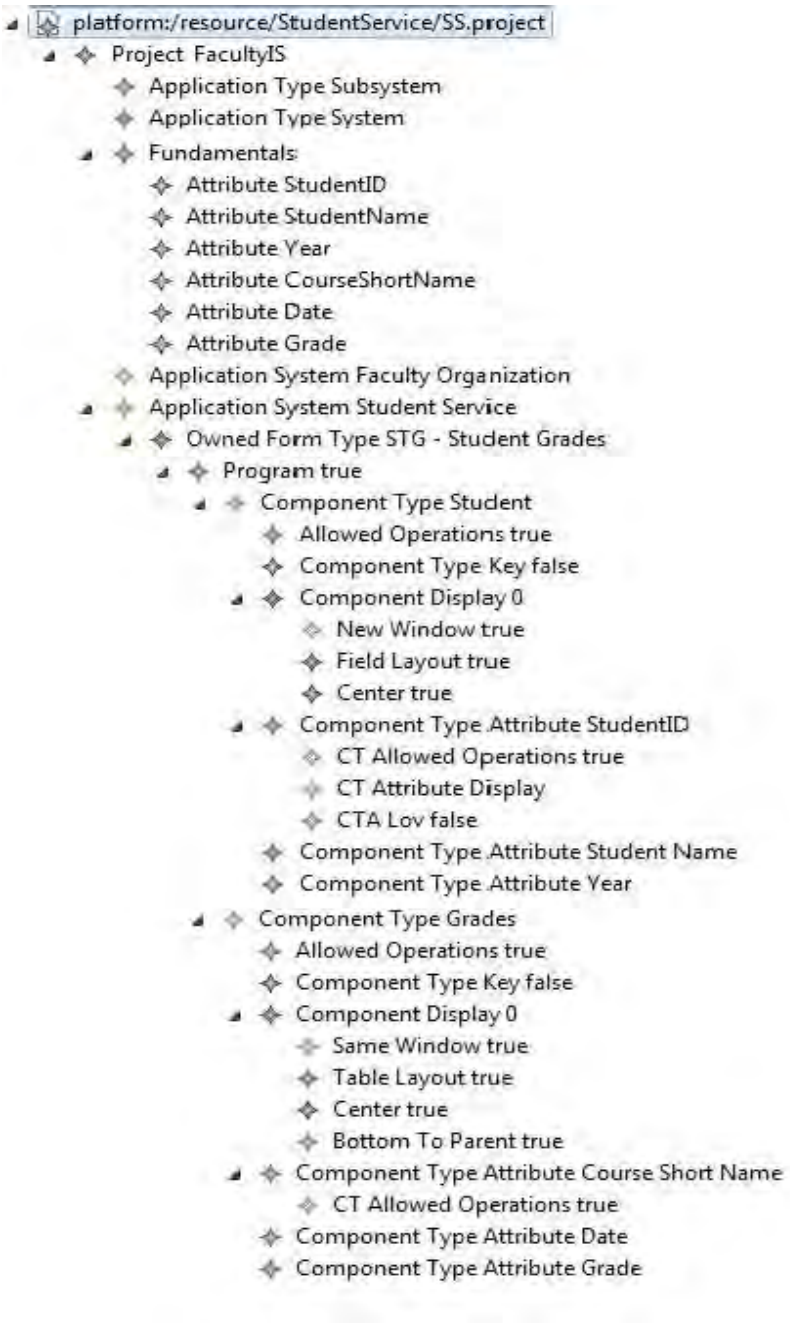

Fig. 6. Model of the Application System Student Service

For each of the *Student* component type attributes, a designer needs to specify its *Name*, *Title*, if it is mandatory or optional for entering values on the

screen form, *Behaviour*, and the list of the *Allowed operations* on the screen form. A set of display and *LOV* properties may also be given. In Fig. 6, at the level of the component type attribute *StudentID*, we presented the properties: *Allowed operations*, *Display* and *LOV*.

In a similar way a designer creates a specification of the *Grades* component type with attributes *CourseShortName*, *Date* and *Grade*. *CourseShortName* is a key of the *Grades* component type.

In this section we have presented an approach to IS conceptual modeling in EMF using IIS*Case PIM concepts. Such approach is valuable not only to create concrete IS models but also to check and validate if IIS*Case PIM concepts are specified correctly and completely. A designer may also use it for fast specification of some IS characteristics. On the other hand, IIS*Case provides specialized, visually oriented and repository based tools supporting the same modeling approach. In general, it is expected to be more convenient for the practical usage, since EMF does not have specialized functionalities and tools to make the IS development process easier for designers.

5. A Concrete Syntax Generation

Generation of the concrete syntax is one of the important steps in the process of the implementation of some DSL. One of our research goals is an implementation of the DSL that will assist in the IS design process. We need to specify the grammar that defines the structure and semantics of the concepts at the meta-level. Such specification actually represents a DSL that could be used in the process of conceptual IS modeling.

A concrete syntax definition is based on the abstract syntax. While concrete syntax expresses a user's perception of a language, the abstract syntax expresses a viewpoint close to the compiler. A DSL implemented for the IIS*Case tool may be used by IS designers. Our plan is to develop a model checker using the abstract syntax specified by EMF. By this, we create a possibility of checking the formal correctness of models, during the whole process of the IS modeling. It is an important feature of each modeling environment aimed at providing IS development in a formal way.

There are different tools for the DSL development. They provide different approaches and techniques to the DSL implementation process. A meta-model specified by Ecore meta-meta-model in EMF may be used as the abstract syntax specification in Eclipse plug-in named EMF text. As we have already developed the meta-model under the EMF using Ecore meta-meta-model, we have decided to use EMF text plug-in and test if the IIS*Case meta-model as the abstract syntax specification may be transformed to the equivalent concrete syntax.

In this section we present only a small part of the concrete syntax grammar, successfully generated by the EMF text plug-in. The IIS*Case meta-model specified by the Ecore meta-meta-model was the input specification for the generation process. The concrete syntax is the output

specification. It is expressed in Human Usable Textual Notation (HUTN) [26] that provides concrete textual language representations for any MOF model. In the following text we present the concrete syntax rules only for the main IIS*Case PIM concepts.

Production rule for defining a *Project* is:

```
Project ::= "Project" "{" "ProjectName" ":"
ProjectName['"','"'] ("NewApplicationType" ":"
NewApplicationType | "NewFundamentalConcept" ":"
NewFundamentalConcept | "NewApplicationSystem" ":"
NewApplicationSystem)* "}";
```

It specifies a name of a project (ProjectName), possible types of application systems (NewApplicationType), different fundamental concepts (NewFundamentalConcept) and application systems (NewApplicationSystem) created in the context of the project.

The rule for specification of an *Application System* is:

```
ApplicationSystem ::= "ApplicationSystem" "{"
"AppSystemName" ":" AppSystemName['"','"']
"AppSystemDescription" ":" AppSystemDescription['"','"']
"AppSystemType" ":" AppSystemType[]
("ParentAppSystem" ":" ParentAppSystem[])?
( "JoinDependency" ":" JoinDependency[] |
"ClosureGraph" ":" ClosureGraph['"','"'] | "BA" ":" BA |
"NewFormType" ":" NewFormType  |
"RelationScheme" ":" RelationScheme[])* "}";
```

It requires specifying the application system name (AppSystemName), description (AppSystemDescription), a type of the application system (AppSystemType), parent application system (ParentAppSystem) created join dependencies (JoinDependecy), a closure graph (ClosureGraph), business applications (BA), form type categories (NewFormType), and generated relation schemes (RelationScheme).

The generated rule for defining *Primitive domain* is:

```
PrimitiveDomain ::= "PrimitiveDomain" "{"
"DomainName" ":" DomainName['"','"']
"Description" ":" Description['"','"']
("DefaultValue" ":" DefaultValue['"','"'])?
("Comment" ":" Comment['"','"'])?
("DecimalPlaces" ":" DecimalPlaces[INTEGER])?
"LenReq" ":" LenReq[] "}";
```

It describes a domain name (DomainName), a description (Description) and a comment (Comment) for the domain, a default value (DefaultValue), decimal places value (DecimalPlaces) and a required length (LenReq).

Production rule for specification a *User defined domain* is:

```
UserDefinedDomain ::= "UserDefinedDomain" "{"
"DomainName" ":" DomainName['"','"']
"Description" ":" Description['"','"']
("DefaultValue" ":" DefaultValue['"','"'])?
("Comment" ":" Comment['"','"'])?
("CheckCondition" ":" CheckCondition['"','"'])?
"USDDT" ":" USDDT[]
"DomainDisplay" ":" DomainDisplay "}";
```

Similar to the previous definition PrimitiveDomain we have DomainName, Description, Comment and DefaultValue. The UserDefinedDomain is also specified by the check condition (CheckCondition), a type of the domain (USDDT) and the specification of how the attributes corresponding to the domain will be displayed (DomainDisplay).

Production rule for defining the *Attribute that is incuded in DB* is:

```
AttributeIncludedInDB ::= "AttributeIncludedInDB" "{"
"AttDomain" ":" AttDomain
"AttributeName" ":" AttributeName['"','"']
"Description" ":" Description['"','"']
"AttDisplay" ":" AttDisplay
"AttIncludedDef" ":" AttIncludedDef
"AttTypeIncluded" ":" AttTypeIncluded "}";
```

The attribute is specified by its name (AttributeName), the attribute domain (AttDomain), a description (Description), the specification of how the attribute is displayed (AttDisplay), a definition of the attribute (AttIncludedDef) and a type of the attribute.

Production rule for the specification of the *Attribute that is not incuded in DB* is similar to the previous one:

```
AttributeNonIncludedInDB ::= "AttributeNonIncludedInDB"
"{" "AttDomain" ":" AttDomain
"AttributeName" ":" AttributeName['"','"']
"Description" ":" Description['"','"']
"AttDisplay" ":" AttDisplay
"AttTypeNonIcluded" ":" AttTypeNonIcluded
"AttNonIncludedDef" ":" AttNonIncludedDef "}";
```

Production rule for definintion of *Function* is:

```
Function ::= "Function" "{"
"FunctionName" ":" FunctionName['"','"']
("Description" ":" Description['"','"'])?
("FunctionBody" ":" FunctionBody['"','"'])?
("FuncParamList" ":" FuncParamList)*
("FunctionReturnType" ":" FunctionReturnType[])? "}";
```

Each function is described by its name (FunctionName), a description (Description), body (FunctionBody), a set of the parameters (FuncParamList) and the function return type (FunctionReturnType).

The specification rule of the *Parameter* is:

```
Parameter ::= "Parameter" "{"
"ParameterSeqNo" ":" ParameterSeqNo[INTEGER]
"ParameterName" ":" ParameterName['"','"']
("ParameterDefValue" ":" ParameterDefValue['"','"'])?
"ParamInOut" ":" ParamInOut
"ParamDomain" ":" ParamDomain[] "}";
```

It requires the definition of a sequence number in the list (ParameterSeqNo), a parameter name (ParameterName), a default value (ParameterDefValue), a type (ParamInOut), and a domain the parameter is corresponding to (ParamDomain)

Production rule for specification of a *Business application* is:

```
BussinesApplication ::= "BussinesApplication" "{"
"BussinesAppName" ":" BussinesAppName['"','"']
"BussinesAppDescription" ":"
BussinesAppDescription['"','"']
("BAEntryFT" ":" BAEntryFT[])* "}";
```

It describes a business application by its name (BussinesAppName), description (BussinesAppDescription), and the entry form type (BAEntryFT).

ReferencedFormType production rule is:

```
ReferencedFormType ::= "ReferencedFormType" "{"
"FormTypeName" ":" FormTypeName['"','"']
("FTCalledFT" ":" FTCalledFT[]
"RefFTAppSys" ":" RefFTAppSys[])* "}";
```

Each referenced form type has its name (FormTypeName), the reference to the called form type (FTCalledFT), and the application system (RefFTAppSys).

Production rule for the definition of an *OwnedFormType* is:

```
OwnedFormType ::= "OwnedFormType" "{"
"FormTypeName" ":" FormTypeName['"','"']
("FTCalledFT" ":" FTCalledFT[])*
"FormTypeTitle" ":" FormTypeTitle['"','"']
("FormTypeFrequency" ":" FormTypeFrequency[INTEGER])?
("FormTypeResponseTime" ":"
FormTypeResponseTime[INTEGER])?
("FTParam" ":" FTParam)*
"DefineFormTypeUsage" ":" DefineFormTypeUsage "}";
```

It requires the definition of a form type specifying its name (FormTypeName), title (FormTypeTitle), the form type that is called (FTCalledFT), frequency (FormTypeFrequency), usage (DefineFormTypeUsage), and the response time (FormTypeResponseTime) of the form type, and the list of the form type parameters (FTParam).

Production rule that represents *Program* definition is:

```
Program ::= selectedFormTypeUsage
["selectedFormTypeUsage" : ""] "Program" "{"
"ConsideredINDBSchDesign" ":" ConsideredINDBSchDesign[]
("NewComponentType" ":" NewComponentType)* "}";
```

The production rule specifies the program by the component type tree structure that consists of a set of component types.

Production rule for the definition of a *Component type* is:

```
ComponentType ::= "ComponentType" "{"
"CompTypeName" ":" CompTypeName ['"','"']
"NoOfOccurrences" ":" NoOfOccurrences['"','"']
"CompTypeTitle" ":" CompTypeTitle['"','"']
"AO" ":" AO  ("IG" ":" IG[])*
("CTU" ":" CTU)*
("CompTypeKey" ":" CompTypeKey)*
("CompTypeCheckConstraint" ":"
CompTypeCheckConstraint['"','"'])?
"CompTypeCompDisplay" ":" CompTypeCompDisplay
("CompTypeParent" ":" CompTypeParent [])?
("CTAttrib" ":" CTAttrib)* "}";
```

ComponentType rule describes a component type specifying its name (CompTypeName), number of occurrences (NoOfOccurrences), a title (CompTypeTitle), allowed operations for the component type (AO), the item group (IG), the unique constraint (CTU) and the key (CompTypeKey).

Production rule for the definition for the *Component type attribute* is:

```
ComponentTypeAttribute ::= CompTypeAttribMandatory
["CompTypeAttribMandatory " : ""]
"ComponentTypeAttribute" "{"
"CompTypeAttribTitle" ":" CompTypeAttribTitle ['"','"']
("CompTypeAttribBehavior" ":" CompTypeAttribBehavior
['"','"'])?
("CTADeafultValue" ":" CTADeafultValue['"','"'])?
"CompTypeAttribName" ":" CompTypeAttribName []
("CTAttribFunction" ":" CTAttribFunction[])?
"CTAttribAO" ":" CTAttribAO "CTAttribLov" ":" CTAttribLov
"CTAttribDisplay" ":" CTAttribDisplay "}";
```

Each component type attribute has its title (CompTypeAttribTitle), behavior specification (CompTypeAttribBehavior), a default value (CTADeafultValue), a reference to the attribute (CompTypeAttribName), a reference to the function (CTAttribFunction), allowed operations for the component type attribute (CTAttribAO), list of values (CTAttribLov), and the set of display properties (CTAttribDisplay).

In Fig. 7 we present a fragment of the program that corresponds to the example specified in Fig. 5.

Firstly, we have created an instance of the project concept, named *Faculty IS.* After that we have specified attributes (AttributeIncludedInDB) with their *AttributeName* values. New attributes are presented in a form of a new fundamental concept instances. Before the specification of an application system, we need to specify one or more application types at the level of the project. In the example shown in Fig. 7, Project *Faculty IS* comprises two application systems (*ApplcationSystem*). While the first one is a specification of the *Faculty Organization* application system, the other one represents *Student Service* application system. For each instance of the *ApplicationSystem* concept it is necessary to define its name (*AppSystemName*), description (*AppSystemDescription*) and type (*AppSystemType*). In Fig. 7, *FacultyOrganization* is a parent (*ParentAppSystem*) application system for the *StudentService.*

At this stage, in the example in Fig. 7, we define a set of the form types (*NewFormType*) for each application system. For each form type, we specify the name (*FormTypeName*), title (*FormTypeTitle)* and the form type usage. Each form type in Fig 7., has the property values for frequency (*FormTypeFrequency*) and response time (*FormTypeResponseTime*). It also includes a list of component type specifications (*NewComponentType*). Form type *STG – Student Grades* comprises two component types, a parent component type (*CompTypeParent*) *STUDENT* and its child component type *GRADES*.

For each component type, in the example presented in Fig. 7, we define the name (*CompTypeName*), title (*CompTypeTitle*) and the set of display properties (*CompTypeCompDisplay*). For *STUDENT* component type search functionality (*SearchFuncionality*) is enabled. The component type *STUDENT* is to be positioned in a new window (*CompDisplayPosition*) and the data need

to be displayed in a field data layout (*CompDispplayDataLayout*). For each component type, we also define component type attributes (*CompTypeAttribute*). The definition of component type attributes requires the name (*CompTypeAttribName)* and the title (*CompTypeAttribTitle*) to be specified.

After the list of component type attributes, the list of component type constraints is given. We may give specifications of key, uniqueness and check constraints. In the example shown in Fig. 7, only component type keys are specified for *STUDENT* by the property *CompTypeKey*.

```
Project {
  ProjectName : "Faculty IS"
  //definition of the fundamental concepts
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "StudentID"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "StudentName"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "Year"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "CourseShortName"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "Date"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "Grade"
    }
  //definition of the applicaiton types
  NewApplicationType :
    ApplicationType {
      ApplicationTypeName : "ProjectSubsystem"
    }
  //definition of the applicaiton systems
  NewApplicationSystem :
    ApplicationSystem {
      AppSystemName : "FacultyOrganization"
      AppSystemDescription : "A unit of a Faculty IS"
      AppSystemType : ProjectSubsystem
```

```
    }
  NewApplicationSystem :
    ApplicationSystem {
      AppSystemName : "StudentService"
      AppSystemDescription : "A unit of a FacultyOrgan."
      AppSystemType : ProjectSubsystem
      ParentAppSystem : FacultyOrganization
      //definition of the new form type
      NewFormType :
        OwnedFormType {
        FormTypeName : "STG-StudentGrades"
        FormTypeTitle : "Catalogue of student grades"
        DefineFormTypeUsage :
          Program {
            ConsideredINDBSchDesign : true
            //definition of the new component type
            NewComponentType :
              ComponentType {
               CompTypeName : "STUDENT"
               CompTypeTitle : "Student Records"
                CompTypeCompDisplay :
                 SearchFuncionality
                 ComponentDisplay {
                  CompDisplayPosition : NewWindow { }
                  CompDispplayDataLayout :FieldLayout { }
                 }
                 CompTypeAttribute :
                  ComponentTypeAttribute {
                    CompTypeAttribName : StudentID
                    CompTypeAttribTitle : "StudentId"
                    CompTypeAttribBehavior : "queryOnly"
                  }
                 CompTypeAttribute :
                  ComponentTypeAttribute {
                    CompTypeAttribName : StudentName
                  }
                 CompTypeAttribute :
                  ComponentTypeAttribute {
                    CompTypeAttribName : Year
                  }
                 CompTypeKey :
                  ComponentTypeKey {
                    CompTypeKeyAttribute : StudentId
                  }
              }
            //definition of the new component type
            NewComponentType :
              ComponentType {
```

```
                CompTypeName : "GRADES"
                CompTypeParent : STUDENT
                  CompTypeCompDisplay :
                   ComponentDisplay {
                    CompDisplayLayoutRelativePosition :
                                          BottomToParent { }
                    CompDispplayDataLayout:TableLayout { }
                   }
      //definition of the new component type continues
              }
          }
          FormTypeFrequency : 1
          FormTypeResponseTime : 1
        }
    }
}
```

Fig. 7. A fragment of program that corresponds to the example in Fig. 5

In this section we presented only a small part of the concrete syntax of a DSL that assists in the process of an IS development. As the process of concrete syntax generation is automatic, we can easily produce a new language based on the whole IIS*Case meta-model. Generated language provides the syntax and semantics for creating the PIM specifications of an IS, which is one of the most important activities in our approach to IS development process.

6. Conclusion

In this paper we presented a part of the IIS*Case PIM meta-model, created by the use of the MOF 2.0 meta-meta model specification. Our intention was not to present all the elements of our meta-model in detail. Instead, we tried to focus just on those meta-model details that are necessary to give a general picture of the model. We believe that the formal specification of our meta-model is not for documentation purposes only. It is also a necessary step in creating a textual DSL to support IS design and give another view of the IS description. In this paper we have presented only one part of the concrete syntax generated from the IIS*Case PIM meta-model. The syntax of such a DSL is not simple. It is a consequence of the complexity of our IIS*Case PIM meta-model. One of the further steps is to generate the whole concrete syntax of the DSL. The concrete syntax should be developed for the textual DSL, although we plan to support the visual approach, too.

The abstract syntax specified by the MOF model is the input specification for the development of the model checker. We may use the IIS*Case PIM meta-model in the verification of generated relational database schemas. Currently, IIS*Case supports an assistance to designers in detecting formal

conflicts at the level of relational database model. By this, the algorithms for detection and resolving constraint collisions at the level of relational data model has already been implemented in IIS*Case. In our future research, we may extend this support so as to assist designers at the level of created PIM models in searching for the appropriate solutions of detected problems. In this way, the process of collision resolving will be raised to the PIM level of abstraction.

Our further research will include experiments with other technologies that rely on MOF. The presented meta-model is a good base for a research in the area of Query View Transform (QVT) set of languages. Our intention is to embed into IIS*Case transformations between different data models. Providing data model transformations may play an important role in the IS design process. In the course of data reengineering process, our plan is to provide the data integration from various sources based on different data models. Data transformation rules specified by QVT could be applied at the level of meta-models specified by various data-models, all expressed in a unified manner in MOF. Our intention is to provide transformations of the models specified in IIS*Case to the UML models. Providing such transformations we allow designers to have models specified in UML standard with OCL constraints.

Acknowledgment. The research presented in this paper was supported by Ministry of Education and Science of Republic of Serbia, Grant III-44010: Intelligent Systems for Software Product Development and Business Support based on Models.

References

1. I. Luković, P. Mogin, J. Pavićević, S. Ristić, “An Approach to Developing Complex Database Schemas Using Form Types”, Software: Practice and Experience, 2007, DOI: 10.1002/spe.820, Vol. 37, No. 15, pp. 1621-1656.
2. I. Luković, M. J. Varanda Pereira, N. Oliveira, D. Cruz, P. R. Henriques, “A DSL for PIM Specifications: Design and Attribute Grammar based Implementation”, Computer Science and Information Systems (ComSIS), ISSN: 1820-0214, DOI: 10.2298/CSIS101229018L, Vol. 8, No. 2, 2011, pp. 379-403.
3. N. Oliveira, M. J. Varanda Pereira, P. R. Henriques, D. Cruz, B. Cramer, “VisualLISA: A Visual Environment to Develop Attribute Grammars”, Computer Science an Information Systems, (ComSIS), ISSN:1820-0214, Vol. 7, No. 2, 2010, pp. 265-289.
4. M. Čeliković, I. Luković, S. Aleksić, V. Ivančević, "A MOF based Meta-Model of IIS*Case PIM Concepts", Federated Conference on Computer Science and Information Systems (FedCSIS), 3rd Workshop on Advances in Programming Languages (WAPL 2011), September 18-21, 2011, Szczecin, Poland, Proceedings, IEEE Computer Society Press and Polish Information Processing Society, ISBN 978-83-60810-39-2, pp. 833-840.
5. Object Management Group (OMG), OCL SpecificationVersion 2.0, [Online] Available: http://www.omg.org/docs/ptc/05-06-06.pdf, June 2005.
6. Eclipse Modeling Framework, [Online] Available: http://www.eclipse.org/ modeling/emf/.

7. Meta-Object Facilty, [Online] Available: http://www.omg.org/mof/.
8. L. Baresi, F. Garzotto, M. Maritati, "W2000 as a MOF Metamodel." In Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics - Web Engineering track. Orlando, USA, 2002.
9. A. Schauerhuber, M. Wimmer, E. Kapsammer, "Bridging existing web modeling languages to model-driven engineering: A metamodel for webML", International Workshop on Model Driven Web Engineering (2nd), Palo Alto, CA, 2006.
10. Document Type definition (DTD), [Online] Available: http://www.w3.org/TR/html4/sgml/dtd.html.
11. M. Richters, M. Gogolla, "A meta-model for OCL" In Proc. of the 2nd international conference on The unified modeling language beyond the standard, ISBN:3-540-66712-1, 1999.
12. F. Jouault, J. Bézivin, "KM3: a DSL for Metamodel Specification", In Proc. of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, 2006, Springer LNCS 4037, pp. 171-185.
13. B. Perišić, G. Milosavljević, I. Dejanović, B. Milosavljević, "UML Profile for Specifying User Interfaces of Business Applications", Computer Science and Information Systems (ComSIS), ISSN: 1820-0214, DOI: 10.2298/CSIS110112010P, Vol. 8, No. 2, 2011, pp. 405-426.
14. Živanov Ž., Rakić P., Hajduković M.: "Using Code Generation Approach in Developing Kiosk Applications", Computer Science and Information Systems, (ComSIS), ISSN:1820-0214, Vol. 5, No. 1, 2008, pp. 41-59.
15. Dejanović I., Milosavljević G., Perišić B., Tumbas M.: A Domain-Specific Language for Defining Static Structure of Database Applications, Computer Science and Information Systems, (ComSIS), ISSN:1820-0214, Vol. 7, No. 3, 2010, pp. 409-440.
16. Van Deursen A, Klint P, Visser J: Domain-specific languages: an annotated bibliography, ACM SIGPLAN Not 35(6), 2000, pp. 26–36.
17. Mernik M., Heering J., Sloane A.M.: When and how to develop domain-specific languages, ACM Computing Surveys, 2005, Vol. 37, No. 4, pp. 316–344.
18. Kos, T., Kosar, T., Knez, J., Mernik, M.: From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer. , Computer Science and Information Systems, (ComSIS), ISSN:1820-0214, Vol. 8, No. 2, 2011, pp. 361-378.
19. GME: Generic Modeling Environment, [Online] Available: http://www.isis.vanderbilt.edu/Projects/gme/.
20. MetaCase Metaedit+, [Online] Available: http://www.metacase.com/.
21. Kelly, S. Lyytinen, K. Rossi,M.: MetaEdit+: a fully configurable multi-user and multi-tool CASE and CAME environment, Advanced Information Systems Engineering 1080, 1996, pp. 1–2.
22. Kelly, S. Tolvanen,J.-P. Domain-Specific Modeling: Enabling Full Code Generation, Wiley–IEEE Computer Society Press, 2008.
23. I. Luković, S. Ristić, P. Mogin, J. Pavićević, "Database Schema Integration Process – A Methodology and Aspects of Its Applying", Novi Sad Journal of Mathematics, Serbia, ISSN: 1450-5444, Vol. 36, No. 1, 2006, pp. 115-150.
24. J. Banović, "An Approach to Generating Executable Software Specifications of an Information System", Ph.D. Thesis, University of Novi Sad, Faculty of Technical Sciences, Novi Sad, 2010.
25. A. Popović, "A Specification of Visual Attributes and Business Application Structures in the IIS*Case Tool", Mr (M.Sc.) Thesis, University of Novi Sad, Faculty of Technical Sciences, 2008.

26. Human Usable Textual Notation (HUTN) [Online] Available http://www.omg.org/spec/HUTN/.

Milan Čeliković graduated in 2009 at the Faculty of Technical Sciences, Novi Sad, at the Department of Computing and Control. Since 2009 he has worked as a teaching assistant at the Faculty of Technical Sciences, Novi Sad, at the Chair for Applied Computer Science. In 2010, he started his Ph.D. studies at the Faculty of Technical Sciences, Novi Sad. His main research interests are focused on: Domain specific modeling, Domain specific languages, Databases and Database management systems. At the moment, he is involved in the projects concerning application of DSLs in the field of software engineering.

Ivan Luković received his M.Sc. (5 year, former Diploma) degree in Informatics from the Faculty of Military and Technical Sciences in Zagreb in 1990. He completed his Mr (2 year) degree at the University of Belgrade, Faculty of Electrical Engineering in 1993, and his Ph.D. at the University of Novi Sad, Faculty of Technical Sciences in 1996. Currently, he works as a Full Professor at the Faculty of Technical Sciences at the University of Novi Sad, where he lectures in several Computer Science and Informatics courses. His research interests are related to Database Systems and Software Engineering. He is the author or coauthor of over 90 papers, 4 books, and 30 industry projects and software solutions in the area.

Slavica Aleksić received her M.Sc. (5 year, former Diploma) degree from Faculty of Technical Sciences in Novi Sad. She completed her Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, she works as a teaching assistant at the Faculty of Technical Sciences at the University of Novi Sad, where she assists in teaching several Computer Science and Informatics courses. Her research interests are related to Database Systems, Theory of Data Models, System Design, Logical and Physical Database Design, Development and Usage of MDSE / CASE tools in Software Engineering and System Design, Reengineering of Information Systems and Model Transformations in MDA.

Vladimir Ivančević is a PhD student in Applied Computer Science and Informatics and a teaching assistant at the Faculty of Technical Sciences, University of Novi Sad (Serbia), where he also gained his BSc and MSc in Electrical Engineering and Computing. His research interests include domain specific languages (DSLs), data mining (DM), and databases. At the moment, he is involved in several projects concerning application of DSLs and DM in the fields of software engineering, education, and public health.

Received: February 03, 2012; Accepted: August 17, 2012.