Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 1611–1620
DOI: 10.15439/2014F239
ACSIS, Vol. 2

Extended Entity-Relationship Approach in a Multi-Paradigm Information System Modeling Tool

Vladimir Dimitrieski, Milan Čeliković, Slavica Aleksić, Sonja Ristić, and Ivan Luković
University of Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
Email: {dimitrieski, milancel, slavica, sdristic, ivan}@uns.ac.rs

***Abstract*—In this paper we present a Multi-Paradigm Information System Modeling Tool (MIST) that supports Extended Entity-Relationship (EER) approach to database design. MIST components currently provide a formal specification of EER database schema specification and its transformation into the relational data model, or the class model. Also, MIST allows generation of Structured Query Language (SQL) code for database creation and procedural code for implementing database constraints. In addition, Java code that stores and processes data from the database, may be generated from the class model.**

I. Introduction

IN THE last few decades, a number of information system (IS) development approaches have emerged. Some of methods that are still in use include: Entity-Relationship (ER) data model proposed by Chen [8] with its further extensions, relational data model [9], form type (FT) model [15], and object-oriented model [22].

Throughout our previous research [2], [15], [16], [17], we have developed a tool named IIS*Studio, to allow a usage of FT concepts in the IS design process. IIS*Studio provides an approach to an evolutive and incremental IS development. The approach is purely platform independent and it strictly differentiates between the specification of a system and its implementation on a particular platform. IIS*Studio currently provides the following functionalities: (i) conceptual modeling of database schemas, transaction programs, and business applications of an IS; (ii) automated design of relational database subschemas in the 3rd normal form (3NF); (iii) automated integration of subschemas into a unified database schema in the 3NF; (iv) automated generation of SQL/DDL code for various DBMSs [2]; (v) conceptual design of common GUI models; (vi) automated generation of executable prototypes of business applications; (vii) modeling check constraints and untypical functionalities of business applications [16]; and (viii) reverse engineering of relational databases to FT models [1].

With the emergence of Model Driven Software Development (MDSD) paradigm and Eclipse Modeling Project (EMP) [10] with an appropriate tooling, we decided to implement some of the existing IIS*Studio functionalities using these technologies. The motivation came from our intention to provide designers of ISs a possibility to use Eclipse environment and thus reduce steep learning curve for new users of IIS*Studio. Therefore, we have developed a domain specific language (DSL) allowing a specification of IS form type models. A detailed specification of this language is presented in [7].

One of the main goals of IIS*Studio is to provide a designer conceptual modeling by creating platform independent models. As designers mainly use other approaches than FT for this purpose, we have decided to support not only FT concepts, but also Extended Entity-Relationship (EER) data model, as a commonly used, traditional approach. Therefore, we have developed a DSL for the specification of EER database schema specifications, named EERDSL. Together, FT and EER are the approaches of our new Eclipse-based tool both providing conceptual database schema modeling. The tool is named Multi-paradigm Information System modeling Tool (MIST). In MIST, both approaches may be used simultaneously. For both FT and EER models, we provide in MIST a transformation into a relational data model. In our previous research on database reengineering approaches [1], we have developed a transformation from a relational data model to a FT model, named relational to form type transformation (R2FT). R2FT is used to transform an EER model into an FT model via relational data model.

As EER approach is present in almost every book on databases, we believe that MIST may also be used for educational purposes, such as learning about: (i) EER concepts and developing a database specification at the conceptual level; (ii) transformations of EER to relational database schema specifications; (iii) transformations of EER to class models; and (iv) MDSD approach by means of the EER approach the students are familiar with, since it is extensively taught in the previous database courses.

In this paper we present the architecture of MIST. It comprises several components that support not only conceptual modeling with FT and EER approaches, but also code generation. The main focus in the paper is on tool components supporting EER approach and transformations from EER to relational and class models. A detailed presentation of the code generators is out of the scope of this paper. Let us just notify here that we have developed both SQL and Java code generators. The SQL Generator provides SQL statements for creating database tables and all basic types of constraints according to SQL ANSI standard. Besides, the code of inverse referential constrains, as they are defined in [3], is generated. As it has to be implemented in a procedural way, different

Research presented in this paper was supported by Ministry of Education, Science and Technological Development of Republic of Serbia, Grant III-44010.

generators are needed for each target database management system (DBMS). Our tool currently supports generation of PL/SQL statements for Oracle DBMS. Our Java code generator provides a generation of Java classes from a class model. Generated Java classes are used in Java programs for storing loaded data from a database.

Apart from Introduction and Conclusion, the paper is organized in four sections. In Section 2 we present the architecture of MIST, while in Section 3 we present EER, Relational and Class meta-models. The aforementioned meta-models are used in the following transformation specifications: (i) the EER data model to relational data model transformation, named EER2Relational and (ii) the EER data model to class model transformation, named EER2Class. These transformations are presented in Section 4. In the same section we present results of applying the aforementioned transformations in our example and the excerpts from generated SQL and Java code. In Section 5, we present related work.

II. The Architecture of MIST

In this section we present the architecture of MIST. Its global picture is depicted in Fig. 1. MIST comprises the following components: FTDSL, Synthesis, Business Application Generator, EERDSL, EER2Rel, EER2Class, SQL Generator, Java Generator, and R2FT. In the following text, we explain each of the components from Fig. 1.

FTDSL component allows designers to specify a platform independent model (PIM) of an IS. FTDSL comprises Ecore meta-model specification of FT PIM concepts and a textual DSL based on these concepts. With the DSL a designer may specify a database schema of an IS, business applications and their graphical user interfaces (GUIs). After an IS is specified at the PIM level, the Synthesis component is used to generate a model of a relational database schema. The Synthesis component implements an improved synthesis algorithm, as it is presented in [14]. First, the component takes a form type specification and transforms it to a Universal Relation Schema (URS) specification. URS and all its benefits are presented in [14]. The synthesis algorithm takes the URS specification and produces a relational database schema as an output. As the FT component may be used to specify business applications of an IS, the MIST architecture includes a Business Application Generator component. This component takes a FT model as an input and generates Java code of a modeled business application. As the specification is enriched with GUI details, the generated application prototype may be executed and used to perform basic CRUD operations over the database.

EERDSL component provides a conceptual specification of an IS database model. Unlike FTDSL, EERDSL is used to specify IS database models only, without specification of business applications and their GUIs. We have created both textual and graphical notations for EERDSL. The textual notation was developed using the Xtext tool, while the Eugenia tool was used to develop the graphical notation. One of the benefits of having a textual notation is that textual editors may be used as an alternative option for more experienced users, or when the Eclipse environment is unavailable. The textual notation also allows a usage of commonly used textual version control systems to provide a better collaboration inside the developer team. Most database designers, however, are using some EER graphical notation. Several different graphical notations for the EER approach exist. Here we have implemented the notation presented by Thalheim in [21]. Both graphical and textual notations may be used by a designer simultaneously, while specifying an EER model. By this, two different viewpoints over the same model are provided in MIST.

EER2Rel component of MIST provides a transformation of EER model to a relational data model. Models being transformed conform to the EER meta-model and relational meta-model, respectively. The meta-models are presented in the following section. The relational data model may be further used in the process of SQL code generation. For this purpose,

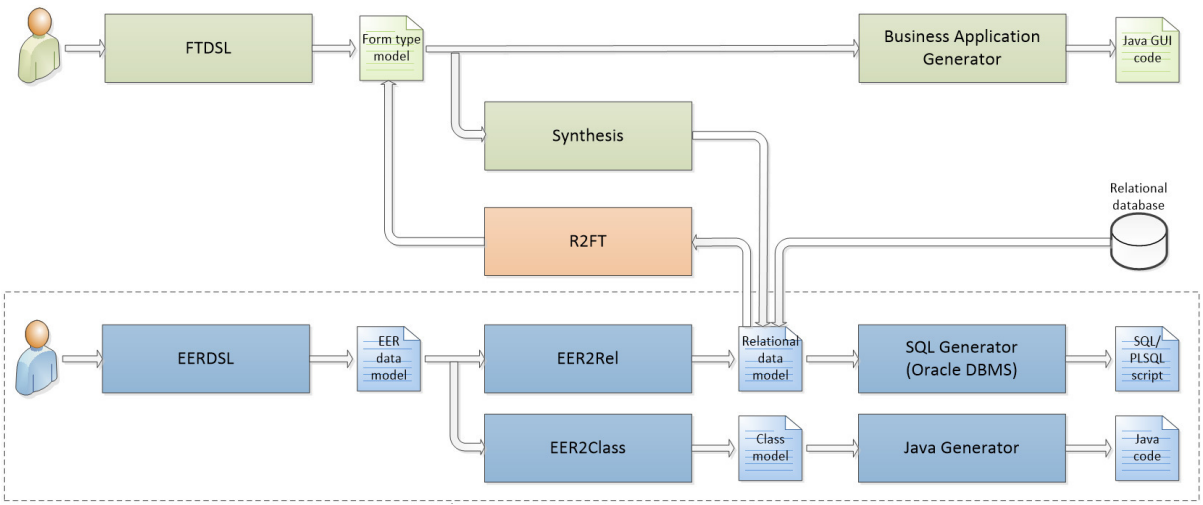

Fig. 1. The architecture of MIST

the SQL Generator component is developed. Currently, it provides generating SQL code for Oracle DBMS.

EER2Class component of MIST provides a transformation of an EER model to a class model. The class model conforms to the class meta-model presented in the Section 3. The class model may be used to generate code in some of the object-oriented programming languages. Our Java Generator component is used to generate Java code from the provided class model.

In order to provide reverse engineering of the relational database model to the FT model, R2FT component has been developed. The component comprises a transformation specification from the relational data model to the FT model. Also, this component may be used to transform an EER data model to the FT model through the relational data model.

III. EER, Relational, and Class Meta-Models

In this section we present in more detail meta-models used in EER approach. In Subsection A, we present our EER meta-model and introduce an example used throughout the paper. Our goal is to test the approach on this example. Further evaluations of the approach are also possible, such as comparing its quality and efficiency with the FT approach. However, this comparison is not presented in this paper due to the space limitations. In Subsection B, we present the relational meta-model, while in Subsection C we present the class meta-model.

A. EER meta-model and an example

In this subsection we present the EER meta-model depicted in Fig. 2. In the rest of this section we present the names of meta-model and model concepts in italic. This meta-model represents the abstract syntax of EERDSL used for specifying data models at the conceptual level. The root concept in our meta-model presented in Fig. 2 is *EERModel*. Each EER model comprises one or more *Entities* and zero or more *Relationships*, *Gerunds*, and *Domains*.

Entity concept is used to specify a class of observed real world entities in the IS being designed. In some approaches, the *Entity* concept is named as *Entity Type* concept. According to [21], we adopt the name *Entity*.

Each entity has zero or more attributes that are modeled by *Attribute* class. Attributes represent properties of real world entities that are of importance for the specified IS. For each attribute, a domain is specified. A domain represents a specification of possible values that can be assigned to an attribute and it is modeled using *Domain*. A domain must be based upon a primitive domain, such as integer, string, real, boolean, date, and time. The primitive domain is modeled by an enumeration *PrimitiveDomain*. An assignment of a domain to an attribute is modeled by *AttributeDomain*. For each attribute, length and default value may be specified. This way of restricting domains allows their reusability. Therefore, domains may be specified once at the level of EER model, and reused and further restricted at the level of attributes. An entity may have one or more keys modeled by *Key*. Each key comprises one or more attributes of the entity. Only one key may be declared as the primary key. This is modeled by *primaryKey* association.

In the meta-model from Fig. 2, different types of relationships between entities are modeled by: *Relationship*, *ISA*, *Categorisation*, and *IdentificationDependency*. An n-ary relationship between entities is modeled by *Relationship*. For each entity its role, minimum, and maximum cardinalities must be specified for each relationship. Minimum cardinality may be provided with the values of zero or one, while a maximum cardinality may be provided with the values of one or more. Entity role and cardinalities are modeled by *RegularEntity*. Each relationship may have zero or more attributes. IS-A

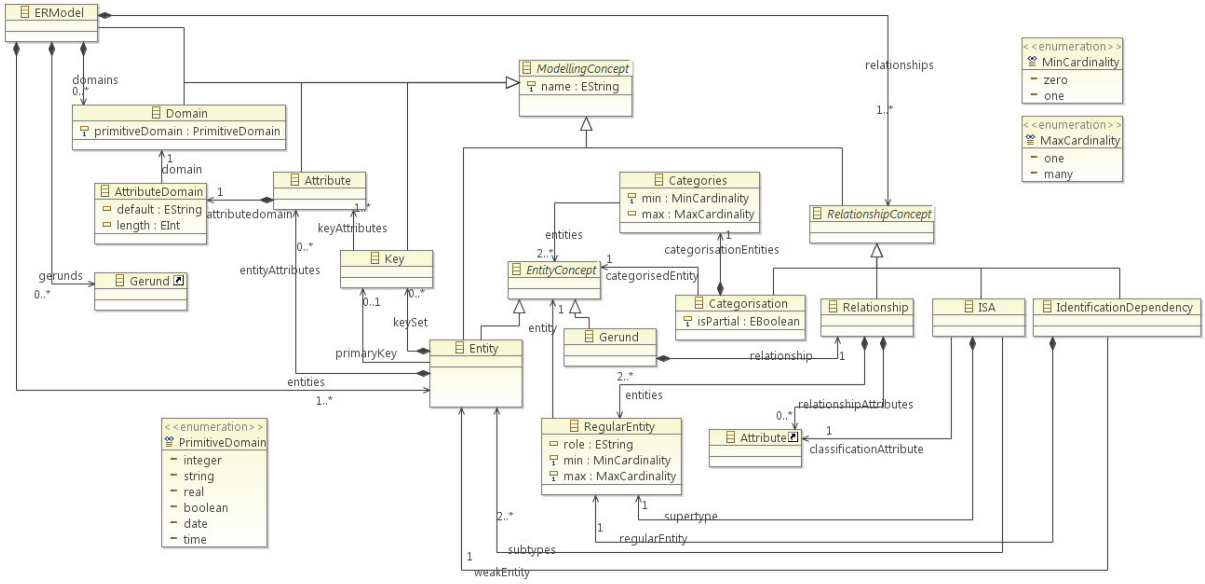

Fig. 2. EER meta-model

relationship, modeled by *ISA*, represents a specialization of entities. For each IS-A relationship a single supertype entity should be provided. This is modeled by *supertype* association. Also, for each IS-A relationship, subtype entities should be provided. This is modeled by *subtype* association. Categorization relationship represents a classification, i.e. typization relationship between entities. It comprises two or more category entities, modeled by *Categories*, and a single categorized entity. Identification dependency relationship concept is modeled by *IdentificationDependency*. A weak entity is modeled by *weakEntity* association, while regular entity is modeled by *regularEntity* association.

An n-ary relationship may participate as an entity in another relationship. Such relationship is called gerund and modeled by *Gerund*. A gerund may take a role of a regular entity in an n-ary or identification dependency relationship. Also, it may take a role of a specialized entity in a specialization, and category entity in a classification.

In Fig. 3, we present our example specified using a textual notation of EERDSL. The textual notation is chosen as it is rarely encountered while specifying EER models. However, due to the limited space, we omit repetitive constructs from the textual specification. We model a part of a Faculty IS, named *FacultySystem*, responsible for storing data about students and their grades.

Students are modeled with an entity named *Student*. Each student has four attributes: *studentID*, *studentsYear*, *studentName*, and *studentSurname*. Primary key named *keyStudent* comprises *studentID* attribute. Subjects are modeled with *Subject* entity having two attributes: *subjectID* and *subjectName*. Primary key named *keySubject* comprises *subjectID* attribute. Teachers of a faculty are modeled as *Teacher* entity and are described by *teacherTitle*, *teacherID*, *teacherName*, and *teacherSurname* attributes. *TeacherID* is the only attribute in *keyTeacher* primary key. Relationship between teachers and subjects they teach is modeled by *TeachesClasses*. Each teacher may teach one or more subjects, while a subject may be taught by one or more teachers. The relation between students and subjects is modeled as *Takes* relationship. Each student may enroll one or more subjects, while a subject may be enrolled by zero or more students. Relationship *Grades* models students' grades given by teachers. As only a teacher that teaches a subject may grade students enrolled on that subject, relationship *Grades* must relate relationships *Takes* and *TeachesClasses*. Therefore, relationships *Takes* and *TeachesClasses* must be represented as gerunds. Each student that takes a subject may be graded by exactly one teacher teaching the subject. A teacher teaching a subject may grade zero or many students of the subject. For each grading *examDate* and *grade* attributes are specified.

```
EERModel FacultySystem {
 domains {
  Domain int primitiveDomain integer,
  ... // omitted domains: varchar, Date, and Time
 }
 entities {
  Entity Student {
   attributeSet {
    Attribute studentName domain varchar(20),
    Attribute studentSurname domain varchar(20),
    Attribute studentID domain int,
    Attribute studentYear domain int
   }
   keySet {
    keyStudent (studentID)
   }
   primaryKey  keyStudent
  },
  ... // omitted entities: Subject and Teacher
 gerunds {
  Gerund Relationship Takes {
   entitiesInRelationship {
    Student (one,many),
    Subject (zero,many)
   }
  },
  ... // omitted gerund for relationship TeachesClasses
 relationships {
  Relationship Grades {
   entitiesInRelationship {
    Takes (one, one),
    TeachesClasses (zero,many)
   }
   attributeSet{
    Attribute grade domain int,
    Attribute examDate domain Date
   }
  }
 }
}
```

Fig. 3. Example of Faculty IS specification in EERDSL

B. Relational meta-model

In this subsection we present our relational meta-model. The root concept of the relational meta-model presented in Fig. 4 is *Database*. Each database schema comprises *Tables*. *SystemDataTypes* represent predefined data types built into each DBMS, while *UserDefinedDataTypes* represent user defined restrictions on existing data types. Each table comprises one or more columns, modeled with *Column*, and constraints inheriting the abstract concept *Constraints*. At the level of a relational database specification, for each table following constraints may be specified: (i) primary key constraint modeled by *PrimaryKeyCon*, comprising an array of columns; (ii) unique constraint modeled by *UniqueCon*, comprising an array of columns having a unique combination of values; (iii) foreign key constraint modeled by *ForeignKey*, comprising: an array of columns, referenced table, and primary key constraint or unique constraint of referenced table; and (iv) check constraint modeled by *CheckCon*, comprising a logical expression.

C. Class meta-model

In this subsection we present our class meta-model. We have modeled only the most basic concepts to specify elements for a representation of data in object oriented programs. The root concept of the class meta-model depicted in Fig. 5 is *ClassModel*. All concepts are grouped into packages modeled by *Package*. A package comprises zero or more *Classes* and

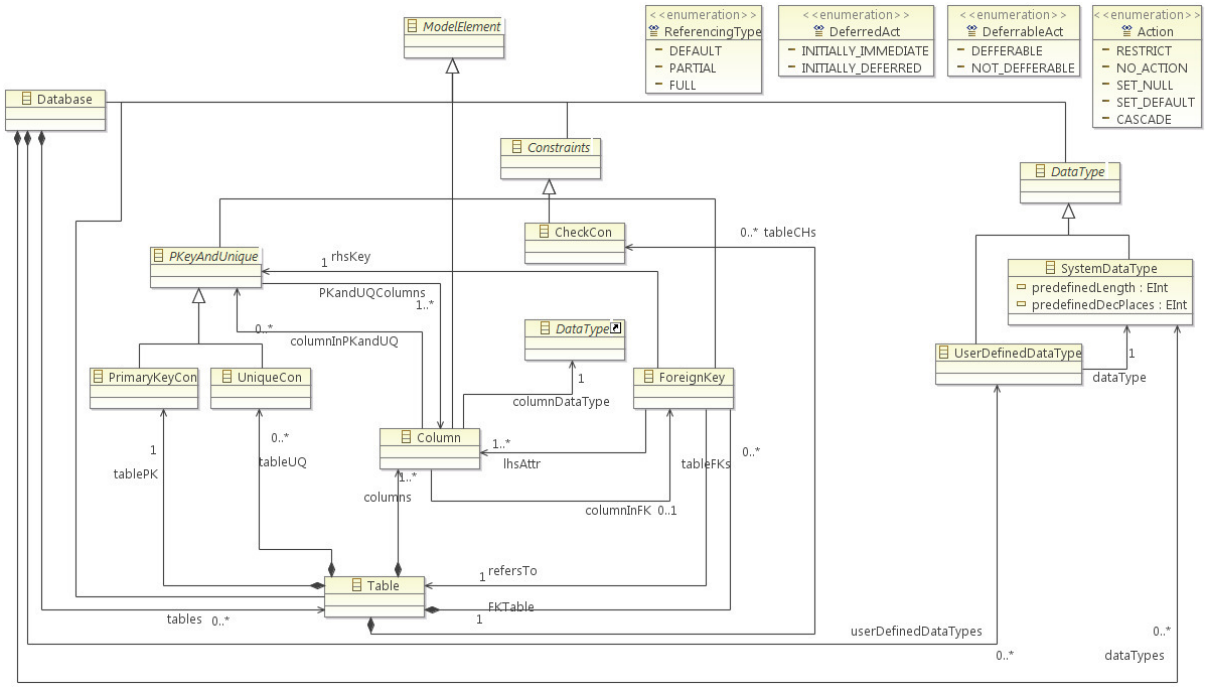

Fig. 4. Relational meta-model

other *Packages*. Each class comprises zero or more fields, modeled by *Attribute* concept, and methods modeled with *Function* concept. Data types are modeled by *Type*. Classes and types inherit the abstract *Classifier*. This concept is specified in order to allow attributes and functions to have both primitive and class types as their type, and the return type respectively. Finally, for each attribute, class, and function an access modifier should be provided. Possible access modifier values are modeled as an enumeration, which comprises following values: private, protected, default, and public.

IV. From EER Data Model to Generated Code

In this section we present two transformations: EER2Rel for transforming an EER model into a relational data model and EER2Class for transforming and EER model into a class model. Transformations are specified in ATL transformation language (ATL) [13]. We present ATL code for the most representative transformation rules. In this section we present the results of applying transformations on the example. Finally, generated code fragments of our example are presented at the end of this section.

Once an EER database model is specified using EERDSL, it may be transformed to the corresponding relational model. In Table I, the first two columns represent all of the corresponding concepts between the EER and relational meta-models. Based on these correspondences, concrete ATL transformation rules are specified.

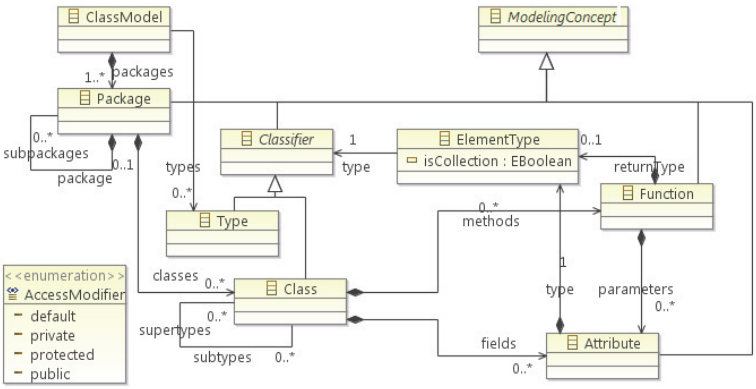

Fig. 5. Class meta-model

TABLE I
TRANSFORMATION ELEMENTS FROM AN EER MODEL TO RELATIONAL AND CLASS MODELS

EER model	Relational model	Class model
EERModel	**Database**	**ClassModel, Package**
Domain	**UserSystemType**, **SystemDataType**	**Type**
Entity	**Table**	**Class**, **Function** (Constructor)
Attribute	**Column**	**Attribute**, **Function** (Get/Set)
Key	**PrimaryKeyCon** (if the key is entity's primary key)	-
Relationship	**Table** (if all maximum cardinalities are *many*)	**Class** (if all maximum cardinalities are *many*), **Function**
Gerund	**Table**	**Class**, **Function**
Identification dependency	**Columns** of the propagated primary key are included into the primary key of the table created from a weak entity. **Foreign key** is created to reference the table created from regular entity.	For a class created from a weak entity, an **object-member** is created referencing a class created from a regular entity.
ISA	**Columns** of the propagated primary key are included into the primary key of the table created from a subtype entity. **Foreign key** is created to reference the table created from supertype entity.	A class created from a subtype entity **inherits** a class created from a supertype entity.
Categorisation	**Foreign key** is created to reference the table created from category entity.	For a class created from a categorized entity, **object-member** is created referencing a class created from a category.

Each EER database concept may be transformed directly to a relational database concept. A system data type is created from EER primitive domains and for each restricted domain, specific user system type is created. Tables are created from three different concepts in EER model: *Entity*, *Gerund*, and *Relationship*. All entities and gerunds are transformed directly into tables, while only the relationships that have maximum cardinality of *many* on both sides, are directly transformed into tables.

In Fig. 6 we present a transformation rule for transforming entities into tables. When transforming an entity to a table, the table name is the same as the name of the entity. For each entity, the following types of columns may be created: (i) columns created from attributes of the entity being transformed; (ii) columns created from attributes of a relationship having a maximum cardinality *one* on the transformed entity side; (iii) columns created from primary key attributes of related entities, where the entity being transformed is on *one* side of a relationship; (iv) columns created from primary key attributes of related categories in categorization relationships; (v) columns created from primary key attributes of related regular entities in identification dependency relationships where transformed entity plays a role of weak entity; and (vi) columns created from primary key attributes of related supertypes in ISA relationships. In this transformation rule, the aforementioned columns are created in both declarative and imperative way. The declarative part of the ATL rule may be used to create the first two types of columns. The creation of these columns does not require the creation of foreign keys and assigning the columns to a foreign key. Therefore, such columns do not require additional references to be set and as such they may be created in a fully declarative way. The imperative section of the ATL rule must be used for the next four types of columns. These columns represent copies of already created primary key columns in other tables. As such, foreign keys must also be created and *lhsAttr* and *columnInFK* references must be assigned to the created foreign keys and columns respectively. A foreign key must also reference a table containing original primary key columns with *refersTo*

```
rule Entity2Table {
  from e : ER!Entity
  using {--omitted variables used in do section }
  to t : RDBMS!Table (
    name <- e.name,
    columns <- e.entityAttributes -> union(
      e.getConnectedRelationshipsForEntity() ->
      collect(e1|e1.relationshipAttributes) ->
      flatten()),
    tablePK <- e.primaryKey
  )
  do {
    --FKs, columns from entities related with Mto1
    --This entity is on one side
    for(r in e.getConnectedRelationshipsForEntity())
    {
      t.tableFKs <- thisModule.CreateForeignKeys(
        r.getConnectedRegularEntity().entity, t,
        r.getConnectedRegularEntity().entity.
          getPrimaryKeyAttributes(),
        r.getConnectedRegularEntity().min=#one,
          r.name);
    }
    --FKs and columns from regular entites in ID rel
    ids <- e.getRegularEntitiesForWeak();
    thisModule.cols <- Sequence{};
    for (reg in ids) {
      t.tableFKs <- thisModule.CreateForeignKeys(
        reg, t, reg.getPrimaryKeyAttributes(),
        false,'ID');
    }
    pkcolumns <- pkcolumns->append(thisModule.cols)
      -> flatten();
    --FKs and columns from supertypes in IS-A rel.
    isaIds <- e.getSupertypes();
    thisModule.cols <- Sequence{};
    for (sup in isaIds) {
      t.tableFKs<-thisModule.CreateForeignKeys(sup,
        t, sup.getPrimaryKeyAttributes(), false,
        'ISA');
    }
    pkcolumns <- pkcolumns->append(thisModule.cols)
      -> flatten();
    --create PK from appropriate columns
    if (t.tablePK.oclIsUndefined()) {
      t.tablePK<-thisModule.Key2PKMtoN(pkcolumns,
        t.name);
    } else {
      t.tablePK.PKandUQColumns <- pkcolumns;
      for (pkc in pkcolumns) {
        pkc.columnInPKandUQ <- t.tablePK;
      }
    }
  }
}
```

Fig. 6. Entity to table transformation

reference. As the created foreign key is a part of the newly created table, the foreign key must be assigned to *FKTable* reference. The creation of these columns and foreign keys is provided in a form of the ATL called rule. We have specified a *CreateForeignKey* called rule which creates columns from primary key columns of a related table and a foreign key referencing that table. This rule adds both created columns and the foreign key to the newly created table and populates all of the aforementioned relationships between foreign key, tables and columns. Due to the space limitations, we have not presented *CreateForeignKey* rule in this paper, but its explicit invocation is presented in Fig. 6 Arguments that are provided are as follows: (i) entity transformed to a table being referenced with a foreign key; (ii) table containing the foreign key, i.e. referencing table; (iii) primary key containing attributes that are transformed to referenced columns; (iv) a boolean value specifying whether an inverse referential integrity should exist; and (v) a string value to be appended to the names of newly created columns in order to avoid name conflicts with previously created columns.

Besides being foreign key columns, columns created from a supertype and a regular entity in ID relationship must be a part of a primary key. In presented ATL rule, as to collect all primary key columns we use a global attribute helper named *cols* and a local variable named *pkcolumns*. *CreateForeignKeys* uses *cols* to return created columns as the return value is used to return a created foreign key. Returned columns are then appended to the other columns contained in *pkcolumns* variable which is local variable declared in the ”using” section of an ATL rule. At the end of an imperative section of *Entity2Table* rule, primary key of a table is set. If a primary key has already been created in the declarative part of the rule, *pkcolumns* are simply appended to *PKandUQColumns* of the existing primary key. The *columnInPKandUQ* relationship is set for each column referencing a primary key the column is a part of. However, a primary key may not be created in the declarative part of the rule. That may be the case if the entity from EER diagram does not have a specified primary key, e.g. a subtype in ISA relationship. For those entities a called rule *Key2PKMtoN* is invoked which creates a primary key and sets all of the appropriate references between columns and the primary key.

Unlike entities, which are all transformed into tables, only relationships that are not contained in gerunds and that have all maximum cardinalities of *many* are transformed into tables. ATL rule that transform such relationships into tables is presented in Fig. 7 Attributes belonging to the relationship are transformed into the table columns. However, when a relationship between two or more entities is transformed from EER to a relational specification it must reference all primary key columns from all related entities. Similarly to the aforementioned *Entity2Table* rule, former columns are created in a declarative way while latter ones are created in an imperative way.

Finally, tables are also created from gerunds. Each gerund encapsulates a single relationship and it may be a part of another relationship. As such, a gerund has all traits of relationships and entities. Therefore, *Gerund2Table* transformation rule is a combination of both *Entity2Table* and *Relationship2Table* rules. It should be noted that a gerund may not be a subtype of inheritance relationship, a weak entity of an identification dependency relationship, or a categorized entity in a categorization relationship. Therefore, the imperative code from *Entity2Table* creating foreign keys and columns from these kinds of relationships is not a part of *Gerund2Table* rule. The code of *Gerund2Table* is not presented in this paper due to the space limitations.

```
rule Relationship2Table {
  from
    --M:N relationship not contained in a gerund
    r : ER!Relationship (not r.isGerund() and
        r.areMaxCardinalitiesMany())
  using {--omitted variables used in do section }
  to
    t : RDBMS!Table (
      name <- r.name,
      columns <- r.relationshipAttributes
    )
  do {
    keys <- r.getConectedKeyAttributesSequence();
    thisModule.cols <- Sequence{};
    for (k in keys) {
      pks <- pks.append(k -> first() ->
        getPKForAttribute());
      t.tableFKs <- thisModule.CreateForeignKeys(
        k -> first() -> getParentConcept(),t,k,
          r.relationship.isIrc(k -> first() ->
            getParentConcept().name)
        k -> first() -> getParentConcept().name);
    }
    t.tablePK <- thisModule.Key2PKMtoN(
      thisModule.cols, t.name);
  }
}
```

Fig. 7. Relationship to table transformation

The transformation of an EER model to a class model is very similar to the *EER2Relational* transformation. In Table I, the first and the third column represent all of the corresponding concepts between the EER and class meta-models. Instead of tables, classes are created and instead of columns each class has attributes. One notable difference is the lack of constraint concepts. In the class model, primary keys are not created which eases the transformation specification. Instead of creation of foreign keys, an attribute of referenced class type is created. If a relationship has a maximum cardinality of *many* then the attribute is a collection of referenced classes. The second difference is the existence of functions. For each class a parametrized constructor is created and for each class attribute, get and set methods are created.

In Fig. 8 we present results of the transformation of our example. In the leftmost part of the figure, we present the same example as in Fig. 3 opened in the Eclipse “Sample Reflective Ecore Model Editor”. This model has served as the input model for both EER2Rel and EER2Class transformations.

Output of EER2Rel is presented in central part of Fig. 8. Each of three EER entities, *Student*, *Teacher*, and *Subject*, has been transformed into a table with the same name.

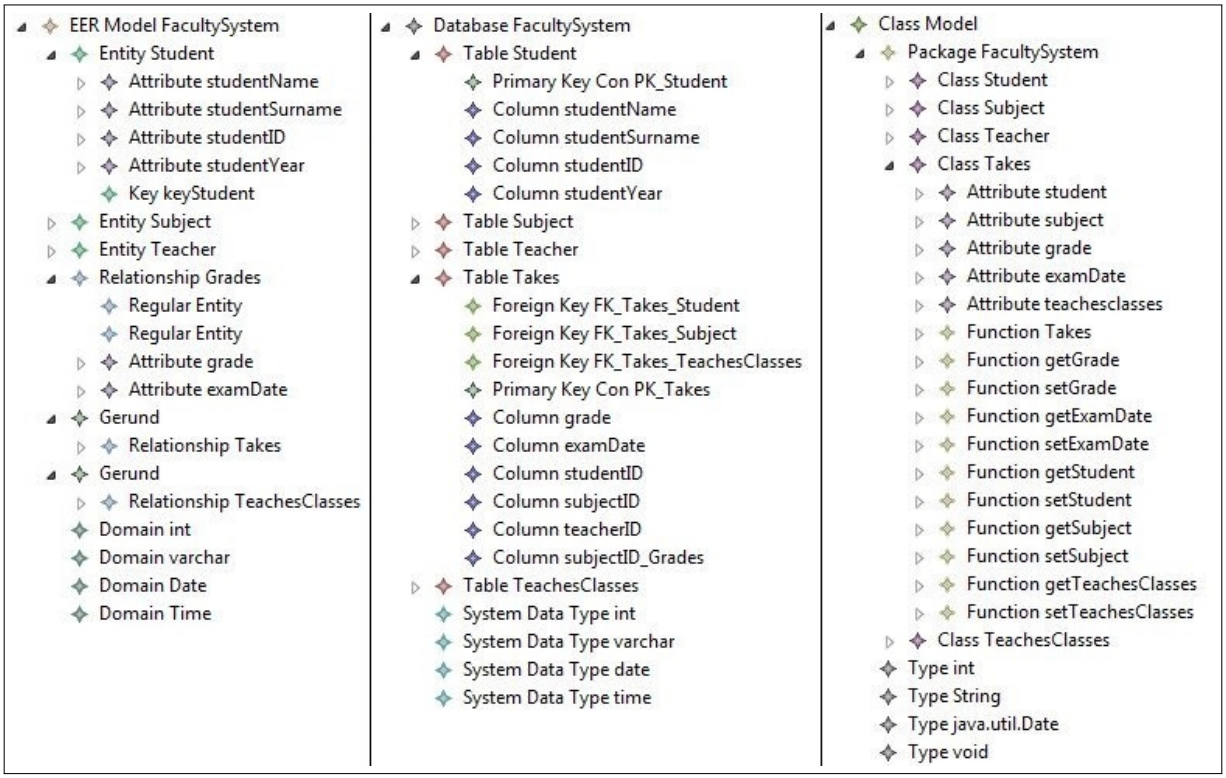

Fig. 8. Faculty IS example: the EER model, the relational database model, and the class model

Here, we present table *Student* in more details. The table contains columns created from *studentName*, *studentSurname*, *studentYear*, and *studentID* attributes of *Student* entity in EER model. From *keyStudent* in *Student* entity, *PK_Student* primary key is created. All the references between the key and attributes are preserved in a form of references between primary key and appropriate columns. Due to a lack of space, these references are not shown in Fig. 8. The gerund *TeachesClasses* is transformed into a table with the same name. Its columns are created from primary key columns of related *Teacher* and *Subject* tables. Appropriate foreign keys are also created for these columns. A primary key of *TeachesClasses* table represents a union of all primary key columns from *Teacher* and *Subject* tables. However, more interesting case of gerund to table transformations is the case of *Takes* gerund. Its relationship relates entities *Student* and *Subject*. Just like in the case of *TeachesClasses* gerund, the resulting table, named *Takes*, will have columns and appropriate foreign keys that reference primary key columns from both *Student* and *Subject* tables. These columns are *studentID* and *subjectID* with their foreign keys *FK_Takes_Student* and *FK_Takes_Subject*, respectively. Gerund *Takes* is related with the gerund *TeachesClasses* with the relationship *Grades*. This relationship is not to be transformed into a separate table as it has a maximum cardinality of *one* on the gerund *Takes* side. Instead, all of the relationship attributes from the *Grades* relationship will be created in *Grades* table as its own columns. Primary key attributes of *TeachesClasses* are to be referenced form appropriate columns in *Takes* table. Therefore, *Takes* table has *grade* and *examnDate* created from the attributes of *Grades* relationship. Columns *teacherID* and *subjectID_grades* with foreign key *FK_Takes_TeachesClasses* reference primary key attributes from *TeachesClasses* table. Let us note that *subjectID* from *TeachesClasses* table had to be renamed in *Takes* table as there was already a column with that name. We chose to append the name of a relationship through which the column was created in the table at the end of a column's name. As names of relationships are unique in the EER model, renamed attributes will all have unique names in their tables.

Output of EER2Class is presented in the rightmost part of Fig. 8. Only a detailed overview of a *Takes* class is given. Similarly to the relational model, a class model has five classes: *Student*, *Subject*, *Teacher*, *Takes*, and *TeachesClasses*. First three classes are created in a straightforward manner from the entities with the same name. *TeachesClasses* is created from *TeachesClasses* relationship. As relationship *Takes* relates *Student* and *Subject* entities, table *Takes* has two object-members: *student* and *subject*. For the same reasons as in the relational model, class *Takes* has two attributes from *Grades* relationship and an object-member representing *Teaches Classes*. These two attributes are grade and *examnDate* and an object-member is named *teachesclasses*. For object-members that represent a collection, a Boolean value of *isCollection* attribute should be set to *true*. For example, object member *teachesclasses* is of

the collection type, as it is created from *Grades* relationship that has a maximum cardinality of *many* on the side of *TeachesClasses*. In addition to attributes, a parametrized constructor and get and set methods are created. For each method a body is automatically generated in a form of a string.

In Fig. 9 we present excerpts from the generated SQL and Java code. In the left part of the figure we present a statement for creating *Takes* table as well as statements that add constraints to this table. In the same figure we present a part of the procedural code that handles inserting new tuples into tables on top of which an inverse referential constraint is enforced. In our example these two tables are *Subject* and *TeachesClasses*. In order to allow simultaneous insert in both tables, a view *View_Subject_TeachesClasses* is created that will allow such insert. An algorithm that handles such insert is a part of a generated trigger named *TRG_IRI_Subject_TeachesClasses_View*. Finally, in the right part of Fig. 9 we present generated Java code for the *Takes* class. We have omitted repetitive code of get and set methods.

V. Related Work

Since Chen proposed ER data model in [8], many papers have been published discussing ER data model, its features, extensions, and practical application. We found only one paper presenting EER data model implementation in the Eclipse environment using MDSD principles. In [12], the authors present EER meta-model and the EERCASE tool based upon it. The tool provides all of the EER concepts represented with Elmasri-Navathe's graphical notation [11].

Our tool is also integrated with the Eclipse environment, so as to provide beginners with an easy of use tool as they are already familiar with the environment. EERDSL component of our tool provides all concepts from the EER approach. All concepts are represented with widely used graphical notation presented also by Thalheim in [21]. Apart from graphical notation, provided by all of the aforementioned tools, our tool also provides EER modeling with a textual notation. Similarly to the aforementioned tools, our tool also supports generation of SQL and Java code from an EER model. Additionally, our tool allows generation of the procedural code for implementation of the inverse referential constraints. Currently, only a generation of PL/SQL code is provided.

There are numerous Computer Aided Software Engineering (CASE) tools to support EER approach, such as PowerDesigner [20], ERWin [5], SmartDraw [19], Oracle Designer [18], or Cameo Data Modeler [6] for MagicDraw. These are mainly commercial and widely used CASE tools and as such they provide proprietary graphical notation for EER usually supporting only selected concepts. In contrast to aforementioned CASE tools, EERDSL provides all of the theoretical EER data modeling concepts. Our tool also supports data modeling using the textual notation. EERDSL is the component of the MIST tool that also provides modeling using the FT concepts. MIST is the only tool that supports the usage of the FT concepts.

VI. Conclusion

During our previous research we developed FT components for our Multi-Paradigm Information System Modeling Tool (MIST). These components provide specification of an IS database schema, business applications and their graphical user

```
...
CREATE TABLE Takes (
  grade int NOT NULL ,
  examDate date NOT NULL ,
  studentID int NOT NULL ,
  subjectID int NOT NULL ,
  teacherID int NOT NULL ,
  subjectID_Grades int NOT NULL
  ,
  CONSTRAINT PK_TAKES PRIMARY KEY (studentID, subjectID)
);

ALTER TABLE Takes ADD (
  CONSTRAINT FK_TAKES_STUDENT FOREIGN KEY (studentID)
    REFERENCES Student (studentID),
  CONSTRAINT FK_TAKES_SUBJECT FOREIGN KEY (subjectID)
    REFERENCES Subject (subjectID),
  CONSTRAINT FK_TAKES_TEACHESCLASSES
    FOREIGN KEY (teacherID, subjectID_Grades)
    REFERENCES TeachesClasses (teacherID, subjectID)
  );

CREATE OR REPLACE VIEW
  View_Subject_TeachesClasses AS ...

CREATE OR REPLACE TRIGGER
  TRG_IRI_Subject_TeachesClasses_View
  INSTEAD OF INSERT ON
    View_Subject_TeachesClasses FOR EACH ROW
DECLARE ... BEGIN ... END;
...
```

```
package facultysystem;

public class Takes {
    protected Student student;
    protected Subject subject;
    private int grade;
    private java.util.Date examDate;
    protected java.util.ArrayList<TeachesClasses> teachesclasses;


    public Takes(Student student, Subject subject,
            int grade, java.util.Date examDate,
            java.util.ArrayList<TeachesClasses> teachesclasses) {
        this.student = student;
        this.subject = subject;
        this.grade = grade;
        this.examDate = examDate;
        this.teachesclasses = teachesclasses;
    };

    public int getGrade() {
        return this.grade;
    };

    public void setGrade(int grade) {
        this.grade = grade;
    };

    //omitted rest of the get/set methods
}
```

Fig. 9. Generated SQL and Java code

interface elements. However, designers widely use EER approach for database schema modeling. Therefore, we provided MIST components supporting EER approach, to offer designers a choice of two alternative conceptual level approaches to create IS specifications at the platform independent level. As both approaches allow a generation of relational database model from a conceptual specification, it is also possible to provide transformations between the two specifications via relational data model. Currently, we have developed a transformation from a relational to the FT model. It allows a designer to create a model using the EER approach, and then use the FT approach to enrich the specification with details of business applications and their GUI elements.

The MIST tool prototype is ready to be used in database and MDSD courses at our faculty. This should provide us with the practical experience and user feedback, allowing further improvement of the tool and new lessons to be learned.

In addition to the conceptual level meta-model and the transformation to the relational data model presented in the paper, we have developed several other model-to-model transformations and code generators. Our SQL Generator component provides generating SQL scripts and procedural code for inverse referential integrity constraints, from a relational model. Also, starting from an EER model, a class model of a database may be created and Java classes are generated. In this paper, our intention was not to give all the details about developed meta-models and transformations. Instead, we tried to focus just on those meta-model details that are necessary to recognize a general picture of the components supporting EER approach.

As components that support EER approach are public to a user and well documented, they may also be used in the educational purposes. It may be used in database courses to assist students in better understanding basic concepts of EER and the rules of EER to relational model transformations. A possible usage is in courses on domain specific languages and model driven software development, as students may familiarize themselves with new concepts using well-known EER concepts.

Several future research directions are possible, including a specification of MIST meta-models semantics and new features of the MIST tool. In order to formally specify semantics of our meta-models, one of the approaches presented in [4] should be used. This could allow us to fully automate the construction of tools supporting our language. Next, in order to fully support simultaneous conceptual specifications with EER and FT approaches, several further research directions are possible. One of them may include an implementation of EER2FT and FT2EER transformations that would allow transformations from one to another conceptual level specification. Also, another research direction would be to extend EERDSL with new concepts allowing more detailed specifications of data models. These new concepts should provide new constraint specifications at the conceptual level. For example, formal specification of database check constraints at the level of EER model is in many approaches poorly supported, or not supported, at all. As we already provide a conceptual specification of the check constraint at the level of FT models, we plan to create the appropriate formalisms for its specification at the level of EER model, too.

REFERENCES

[1] S. Aleksic, "Methods of database schema transformations in support of the information system reengineering process," Ph.D. dissertation, University of Novi Sad, 2013.

[2] S. Aleksic, I. Lukovic, P. Mogin, and M. Govedarica, "A generator of SQL schema specifications," *Computer Science and Information Systems*, 2007. [Online]. Available: http://dx.doi.org/10.2298/CSIS0702081A

[3] S. Aleksic, S. Ristic, I. Lukovic, and M. Celikovic, "A design specification and a server implementation of the inverse referential integrity constraints," *Computer Science and Information Systems*, 2013. [Online]. Available: http://dx.doi.org/10.2298/CSIS111102003A

[4] B. R. Bryant, J. Gray, M. Mernik, P. J. Clarke, R. B. France, and G. Karsai, "Challenges and directions in formalizing the semantics of modeling languages," *Computer Science and Information Systems*, 2011. [Online]. Available: http://dx.doi.org/10.2298/CSIS110114012B

[5] "CA ERwin." [Online]. Available: http://erwin.com/

[6] "Cameo Data Modeler." [Online]. Available: http://www.nomagic.com/products/magicdraw-addons/cameo-data-modeler.html

[7] M. Celikovic, I. Lukovic, S. Aleksic, and V. Ivancevic, "A MOF based meta-model and a concrete DSL syntax of IIS*Case PIM concepts," *Computer Science and Information Systems*, 2012. [Online]. Available: http://dx.doi.org/10.2298/CSIS120203034C

[8] P. P.-S. Chen, "The entity-relationship model toward a unified view of data," *ACM Transactions on Database Systems*, 1976. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-59412-0_18

[9] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, 1970. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-59412-0_16

[10] "Eclipse Modeling Project (EMP)." [Online]. Available: http://projects.eclipse.org/projects/modeling

[11] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Addison-Wesley, 2010.

[12] R. N. Fidalgo, E. Alves, S. Espana, J. Castro, and O. Pastor, "Metamodeling the enhanced entity-relationship model," *Journal of Information and Data Management*, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34002-4_40

[13] F. Jouault, F. Allilaire, J. Bezivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.08.002

[14] I. Lukovic, "From the synthesis algorithm to the model driven transformations in database design," in *Proceedings of 10th International Scientific Conference on Informatics (Informatics 2009), Herlany, Slovakia*, 2009.

[15] I. Lukovic, P. Mogin, J. Pavicevic, and S. Ristic, "An approach to developing complex database schemas using form types," *Software: Practice and Experience*, 2007. [Online]. Available: http://dx.doi.org/10.1002/spe.820

[16] I. Lukovic, A. Popovic, J. Mostic, and S. Ristic, "A tool for modeling form type check constraints and complex functionalities of business applications," *Computer Science and Information Systems*, 2010. [Online]. Available: http://dx.doi.org/10.2298/CSIS1002359L

[17] I. Lukovic, S. Ristic, P. Mogin, and J. Pavicevic, "Database schema integration process a methodology and aspects of its applying," *Novi Sad Journal of Mathematics*, 2006.

[18] "Oracle Designer." [Online]. Available: http://www.oracle.com/technetwork/developer-tools/designer/overview/index-082236.html

[19] "SmartDraw." [Online]. Available: http://www.smartdraw.com/

[20] "Sybase PowerDesigner." [Online]. Available: http://www.sybase.com/products/modelingdevelopment/powerdesigner

[21] B. Thalheim, *Entity-relationship modeling: foundations of database technology*. Springer, 2000.

[22] R. S. Wazlawick, *Object-oriented analysis and design for information systems*. Morgan Kaufmann, 2014.