Computer Science and Information Systems 15(3):821–843 https://doi.org/10.2298/CSIS180324029V

Extended Tuple Constraint Type as a Complex Integrity Constraint Type in XML Data Model – Definition and Enforcement*

Jovana Vidaković[1], Sonja Ristić[2], Slavica Kordić[3], Ivan Luković[3]

[1] University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics
Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia
[2]University of Novi Sad, Faculty of Technical Sciences, Department of Industrial Engineering and Engineering Management
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
[3]University of Novi Sad, Faculty of Technical Sciences, Department of Computing and Control
Trg Dositeja Obradovića 6, 21000 Novi Sad, Serbia
{jovana, sdristic, slavica, ivan}@uns.ac.rs

Abstract. A database management system (DBMS) is based on a data model whose concepts are used to express a database schema. Each data model has a specific set of integrity constraint types. There are integrity constraint types, such as key constraint, unique constraint and foreign key constraint that are supported by most DBMSs. Other, more complex constraint types are difficult to express and enforce and are mostly completely disregarded by actual DBMSs. The users have to manage those using custom procedures or triggers. eXtended Markup Language (XML) has become the universal format for representing and exchanging data. Very often XML data are generated from relational databases and exported to a target application or another database. In this context, integrity constraints play the essential role in preserving the original semantics of data. Integrity constraints have been extensively studied in the relational data model. Mechanisms provided by XML schema languages rely on a simple form of constraints that is sufficient neither for expressing semantic constraints commonly found in databases nor for expressing more complex constraints induced by the business rules of the system under study. In this paper we present a classification of constraint types in relational data model, discuss possible declarative mechanisms for their specification and enforcement in the XML data model, and illustrate our approach to the definition and enforcement of complex constraint types in the XML data model on the example of extended tuple constraint type.

Keywords: XML Data Model, extended tuple constraint, code generation, XQuery functions, database triggers.

* This paper is the extended version of the paper [1]

1. Introduction

Data quality is essential for organizations as they try to derive value from data. Integrity constraints are mechanisms for ensuring data consistency, a very important aspect of data quality. A data model provides the means to achieve data abstraction and to express a database schema. The set of supported integrity constraint types is one of the data model's elements. Some of constraint types are common for several data models and some of them are specific only for a certain data model. A database management system should have mechanisms for defining and enforcing integrity constraints.

The relational data model is a superior logical data model and the acceptance of relational DBMSs (RDBMS) is widespread, too. RDBMSs have support for many different types of constraints. Some of them are well-known like domain constraints, key and unique constraints, NULL value constraints and referential integrity constraints. These constraints are called built-in constraints. The complexity of real world systems imposes the need for more complex integrity constraint types. They are very often difficult to define, implement and enforce. Integrity constraint specifications are translated into constraint enforcing mechanisms provided by the DBMS used to implement a database. Built-in constraints can be implemented by declarative RDBMSs mechanisms. It means that they can be specified within CREATE/ALTER TABLE statement and thereafter RDBMSs automatically enforce them. The declarative implementation of more complex constraint types is not supported by RDBMSs. Most of the contemporary RDBMSs offer efficient procedural support for constraint implementation by means of triggers. Triggers are procedural mechanisms to specify automatic actions that a DBMS will perform when certain events and conditions occur. This implies an excessive programmers' effort to maintain integrity and develop applications.

The eXtensible Markup Language (XML) is a standard widely used for data representation and interchange. Accompanied with a stack of languages and tools, such as XML Schema Definition (XSD), XPath and XQuery e.g., it becomes a powerful technology. XML documents are very often used for data transport between different databases; between databases and programs; and between different programs. That phenomena caused the need to store, process and query XML documents, resulting in DBMSs based on XML technologies. In recent years, XML databases, that have an XML document as their fundamental unit of logical storage, have been widely in use. They provide a full range of core database services like persistent storage, ACID (Atomic, Consistent, Isolated, and Durable) transactions and security. XML schema definition languages and concepts can be used to represent a set of data types, a set of data structure types, a set of data operation types, and a set of integrity constraint types. The four-tuple of these sets represents XML data model. XML DBMSs offer specification and implementation of several integrity constraints, like keys, foreign keys and unique constraints. Other constraints that may exist in a large database project are not supported in XML DBMSs. That is the reason why these kinds of constraints are ignored by database designers in the way that they do not recognize, specify and implement them by means of XML DBMSs.

In one of our previous papers [2] we have proposed the constraint taxonomy in XML databases. The proposed taxonomy relies on the constraint taxonomy in relational data model. Our decision to rely XML taxonomy on relational data model was motivated by the fact that in many applications, XML data is generated from relational databases, or

exported to a relational database or target application. In relational database, integrity constraints like attribute value constraints, key constraints and referential integrity constraints (also called foreign key constraints) convey a fundamental part of the information. In this context, integrity constraints play an essential role in preserving the original semantics of data.

In this paper we propose a classification of integrity constraint types in the relational data model in order to determine more closely the set of complex integrity types. The classification is made to generalize results presented in [3] and [1]. In [3] we have explained the need for introducing the extended referential integrity constraint (ERIC) type, as a complex constraint type, in the relational and XML data model. We also specified the ERIC type in the XML data model and proposed two techniques for implementation of ERIC in XML DBMSs. In [1] we have expanded our research on the extended tuple constraint (ETC) type that can also be characterized as a complex integrity constraint type. We defined the ETC in the relational data model, described two ways for its implementation in RDBMSs, and proposed ETC's specification and validation in the XML data model. This paper is the extended version of the paper [1]. Here we use ETC type to illustrate how the constraint type characteristics given in [2] can be applied to define ETC by means of the XML data model and how to enforce it in an XML database. We extend [1] with the explanation of the extended XML Schema and the actions in case of violation of the constraint. New sections 5, 6 and 7 are added and they contain the description of the code generator, which can transform constraint specifications into error free XQuery functions or triggers for constraint validation in the XML data model.

In that way, we would like to enhance the usage of more complex constraint types (ETC type in particular) in the database design theory and practice. The existence of the automated code generator for ETCs validation in XML database would motivate designers to identify and specify ETCs in the designed database schema.

Apart from Introduction, this paper is organized as follows. The basic notions about the data models and constraint types are given in Section 2. In Section 3 we propose a classification of the integrity constraint types in the relational data model and give the explanation of the ETC type. The extended tuple constraint in the XML data model is defined in Section 4, and the architecture of the code generator for constraint validation in XML databases is proposed in Section 5. In sections 6 and 7 we present the code generator for ETC validation in XML databases as well as the usage of the generated code for validation of the extended tuple constraint. Related work is discussed in Section 8 and Section 9 contains the conclusion and future work guidelines.

2. Preliminaries

In this chapter we introduce some preliminaries that will be used in the rest of the paper.

A database is a collection of related data stored on a storage medium controlled by a database management system (DBMS). The description of the database that is specified during database design is called **database schema**. A **data model** provides the means to achieve data abstraction and to a express database schema. According to Date and Darwen definition [4] revised by Eessaar [5]: "A data model is an abstract, self-contained, implementation-independent definition of elements of a 4-tuple of sets (*T, S,*

O, *C*) that together make up the abstract machine with which database users interact, where *T* is a set of data types; *S* is a set of data structure types; *O* is a set of data operation types; *C* is a set of integrity constraint types." Numerous data models, like Entity-Relationship (ER), relational or XML data models are proposed with different *T*, *S*, *O* and *C* sets. For example, let C_{ER} denote the set of integrity constraint types from ER data model and C_{REL} denote the set of integrity constraint types from relational data model. These two sets are different. The cardinality constraint type belongs to C_{ER} but it does not belong to C_{REL}. At the same time, the referential integrity constraint type belongs to C_{REL} and does not belong to C_{ER}.

Here we focus on the relational and XML data models and their sets of integrity constraint types.

Elmasri and Navathe [6] divide constraints on databases into three main categories: i) constraints that are inherent in the data model i.e. implicit constraints; ii) constraints that can be expressed in database schemas of the data model by means of data definition language (DDL) i.e. explicit constraints; and iii) constraints that must be expressed and enforced procedurally by the application program i.e. semantic constraints. Semantic constraints are instances of complex constraint types.

In the paper we deal with the ETC type as the representative of the set of complex constraint types. ETC type is the extension of a tuple constraint type of relational data model. A tuple constraint is defined in the context of only one relation scheme while as an ETC is defined in the context of two or more relation schemes and corresponding relations. The following two constraints are examples of a tuple constraint and an ETC, respectively: i) *the value of employee's monthly credit installment cannot be greater than the third of his monthly salary*; and ii) *monthly salary of an employee should not exceed the salary of the employee's supervisor*. Detail description of ETC type is given in Section 3.

3. Classification of Integrity Constraint Types in Relational Data Model

Integrity constraints have been proved as fundamentally important in the database management. Their extensive study through decades of development and application of relational data model shows that reasoning about constraints is a non-trivial task and that even simple constraint languages can have high complexity. In this section we propose the classification of integrity constraint types in the relational data model.

In this paper we use some basic notions of relational data model that are reused, among other sources, from [4] and [6] and presented in [7].

Let *R* be a finite set of attributes. For each attribute $A \in R$, the set of all its possible values is called the domain of *A*. The domain associated with an attribute *A* is denoted by *Dom*(*A*), and the set of possible values of attribute *A* (*A*-values) is denoted by *dom*(*A*). A domain constraint restricts allowed values within a certain domain. A tuple *t* over $R = \{A_1, ..., A_m\}$ is a sequence of values $(a_1, ..., a_m)$ where: $(\forall i \in \{1, \ldots, m\})(a_i \in dom(A_i))$. A relation over *R*, denoted with *r*(*R*), is a set of tuples over *R*.

Relational database schema describes data stored in a database. Formally, a relational database schema is an order pair (*S*, *I*), where *S* is a finite set of relation schemes and *I* a finite set of multiple relational constraints. A relation scheme is a named order pair *N*(*R*,

C) where *N* is the name of relation scheme, *R* is a finite set of attributes and *C* a finite set of relational constraints. *C* contains attribute value constraints alongside with null constraints, tuple constraints, unique constraints and key constraints. A set of multiple relational constraints *I*, contains extended tuple constraints and inclusion dependencies.

In Fig. 1 the classification of integrity constraint types in relational data model is presented.

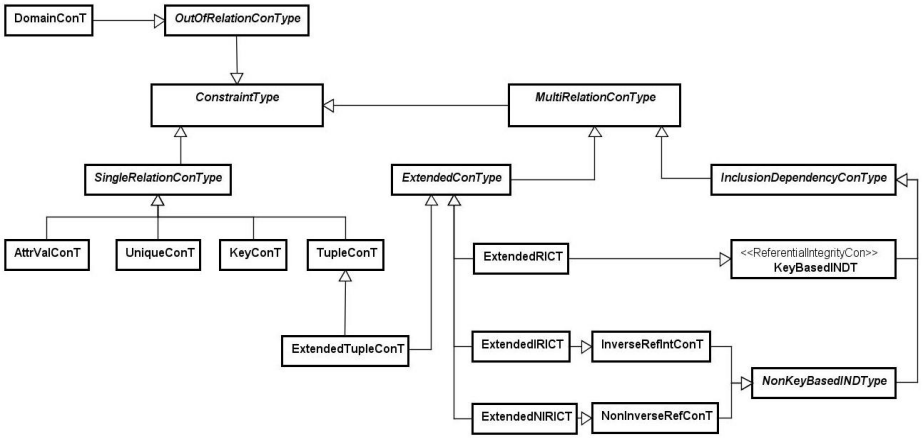

Fig. 1. The classification of integrity constraint types in relational data model

Integrity constraint types can be classified in three classes depending on the definition scope: out of relation constraint types (OutOfRelationConType), single relation constraint types (SingleRelationConType) and multi relation constraint types (MultiRelationConType). Domain constraint type (DomainConT) is an example of out of relation constraint type, because it is defined without reference on any relation scheme within a relational database schema. Attribute value constraint type alongside with null constraint (AttrValConT), unique constraint type (UniqueConT), key constraint type (KeyConT) and tuple constraint type (TupleConT) are classes of a single relation constraint type since they are defined within the context of one relation scheme.

The multi relation constraint type may be specialized into two classes: inclusion dependency constraint type (InclusionDependencyConType) and extended constraint type (ExtendedConType). They are defined within the scope of more than one, not necessarily different, relation schemes. All multi relation constraint types, except referential integrity type (KeyBasedINDT), are complex integrity types. Inclusion dependency constraint type and its specializations (referential integrity constraint type, inverse referential integrity constraint type, non-inverse referential integrity type, extended referential integrity constraint type, extended inverse referential integrity constraint type and extended non-inverse referential integrity type) are not in the focus of this paper. Consequently, they will not be described here. Details about inclusion dependency constraint type and its specializations alongside with examples that illustrate their application can be found in [7, 8]. The extended referential integrity constraint type (ExtendedRICT) explanation and its specification in the XML data model and implementation in XML DBMSs are given in [3]. The extended constraint type is an abstract constraint type that models multi relation constraint types that are interpreted over a join between more than one, not necessarily different, relations. In

general, the join may be a theta-join (Θ-join), equijoin or natural join. Here we focus just on the extended tuple constraints as one of its specializations. In the following text we will explain the tuple constraint type and extended tuple constraint type that inherits characteristics of both tuple constraint type and extended constraint type.

For the specification of a relation scheme we use shortened notation $N(R, K_p)$ where N is the name of relation scheme, R is a finite set of attributes and K_p is a primary key. A relation over R, denoted with $r(R)$, is a set of tuples over R. We will use the term "relation N" for the relation $r(R)$ that is a relation over the set of attributes R of the relation scheme $N(R, K_p)$.

Definition 1. The tuple constraint is a relational constraint type defined over a set of attributes of a single relation scheme. The syntax of the formula for specifying a tuple constraint is:

$$\tau(N) = (\{ \tau(N, A) \mid A \in R\}, Con(N)),$$

where $\tau(N)$ is the tuple constraint defined over the relation scheme $N(R, K_p)$, $\tau(N, A)$ is the attribute value constraint, and $Con(N)$ is the logical condition of the tuple constraint. The logical condition is defined over at least two attributes from a single attribute set R. The tuple constraint is interpreted for each tuple t that belongs to a relation N. □

The Example 1 illustrates the usage of tuple constraint. The following examples are also given in [1].

Example 1. Let the relation scheme *Employee* model employees of a company:

Employee({*EmpId*, *FirstName*, *LastName*, *ManagerId*, *ManagDate*, *BirthDate*, *EmploymentDate*}, {*EmpId*}).

According to the attribute value constraints τ(*Employee*, *BirthDate*) and τ(*Employee*, *EmploymentDate*), values of attributes *BirthDate* and *EmploymentDate* in a tuple from a relation *Employee* should be valid dates and must not contain NULL values. Besides, there is a mutual conditionality between the values of *BirthDate* and *EmploymentDate* in a single tuple. An employee must be at least 16 years old at the beginning of the year of employment. This constraint cannot be specified by means of an attribute value constraint. It can be formally expressed by means of a logical condition:

$$Con(N): year(EmploymentDate) - year(BirthDate) > 15. \square$$

A tuple constraint is defined in the context of one relation scheme. It cannot be used to mutually constrain values of attributes which belong to different attribute sets. The different attribute sets may be the attribute sets of different relation schemes (Example 2), or they may be attribute sets of a single relation scheme that appears in different roles (Example 3). Extended tuple constraint is used to specify such kind of data constraint. The Example 2 and Example 3 illustrate the usage of ETC.

Example 2. Let the relational database schema model persons and their documents like ID card, driving license, and passport. The relation database schema contains two different relation schemes: *Person* and *Document*:

$$Person(\{PersId, FirstName, LastName, Gender, BirthDate\}, \{PersId\}); \text{ and}$$
$$Document(\{DocNum, DocType, IssueDate, PersId\}, \{DocId\}).$$

The relation scheme *Person* has a primary key *PersId* and the relation scheme *Document* has a primary key *DocNum*. The attribute *PersId* is a foreign key in the relation scheme *Document* which is specified with the referential integrity constraint $Document[PersId] \subseteq Person[PersId]$.

There is a mutual conditionality between the values of the attributes *BirthDate* and *IssueDate*. The issue date of a document must be greater than or equal to the birth date of a person to whom that document belongs. This real word constraint can be expressed by an extended tuple constraint. It is extended since it constrains values of the attributes from two different relations. At the same time, it is a tuple constraint since these values are from only one tuple that belongs to a join (in this example it is a natural join) of two relations. This constraint can be formally expressed as follows:

$$\tau_{ex}(Person \triangleright\triangleleft Document) = Con(Person \triangleright\triangleleft Document): IssueDate \geq BirthDate.$$

There are several database operations that could violate the aforementioned ETC: i) tuple insert in the relation *Document*; ii) tuple update in the relation *Document* that changes values of attributes *IssueDate* or *PersId*; and iii) tuple update in the relation *Person* that changes values of attributes *BirthDate* (under the assumption that values of primary key cannot be changed). Implementation of the ETC would support constraint validation in all of these cases. □

Example 3. Let us consider relation scheme presented in Example 1. *ManagerId* is a foreign key in the relation scheme *Employee*, which is specified with the referential integrity constraint $Employee[ManagerId] \subseteq Employee[PersId]$. It models relationships between the employees and theirs direct superior managers (one direct superior manager per an employee). The attribute *ManagDate* in a tuple *t* represents the day from which the employee, represented with tuple t_1, has got the manager with the actual $t_1(ManagerId)$ value. The value of *ManagDate* must be greater than or equal to the value of *EmploymentDate* of that manager. This value is stored in the tuple t_2 of the *Employee* relation, that obeys the following condition: $t_2(EmpId) = t_1(ManagerId)$. The *Employee* relation has two roles: $Employee_{Emp}$ role models all employees, and $Employee_{Man}$ role models all managers. The aforementioned ETC is formally expressed as follows:

$$\tau_{ex}(Employee_{Emp} \triangleright\triangleleft Employee_{Man}) = Con(Employee_{Emp} \triangleright\triangleleft Employee_{Man}):$$
$$ManagDate_{Emp} \geq EmploymentDate_{Man}.$$

The join between relations is not a natural, but an equijoin with the join condition that equals value of attribute *ManagerId* from $Employee_{Emp}$ role and value of attribute *EmpId* from $Employee_{Man}$ role. □

The three presented examples illustrate the importance of an efficient recognition, specification and implementation of complex constraints.

The universal syntax of the formula for specifying an extended tuple constraint is given by the expression:

$$\tau_{ex}(N_1 \triangleright\triangleleft \ldots \triangleright\triangleleft N_m) = Con(N_1 \triangleright\triangleleft \ldots \triangleright\triangleleft N_m),$$

where $\tau_{ex}(N_1 \triangleright\triangleleft \ldots \triangleright\triangleleft N_m)$ is the extended tuple constraint over the relation schemes $N_1(R_1, K_{p1})$, ..., $N_m(R_m, K_{pm})$, and $Con(N_1 \triangleright\triangleleft \ldots \triangleright\triangleleft N_m)$ is the logical condition over a subset S of union of attribute sets $R_1,\ldots, R_m$. Relation schemes $N_1,\ldots, N_m$ do not need to be different. This constraint is validated for each tuple t that belongs to the join of relations $N_1,\ldots, N_m$ respectively:

$$t \in r(R_1) \triangleright\triangleleft \ldots \triangleright\triangleleft r(R_m),\ \tau_{ex}(N_1 \triangleright\triangleleft \ldots \triangleright\triangleleft N_m)(t) = Con(N_1 \triangleright\triangleleft \ldots \triangleright\triangleleft N_m)(t).$$

Critical operations that can violate an ETC are tuple insert into relations $N_1,\ldots, N_m$ and tuple update that changes values of attributes from the subset S.

Most of the contemporary RDBMSs offer declarative definition and implementation of domain, unique, key, tuple and referential integrity constraints. The implementation of constraints of other types presented in Fig. 1 is supported only by the procedural mechanisms.

In the current XML specifications, XML Schema definitions include type definitions, occurrence cardinalities, unique constraints, and referential integrity. In this paper the discussion about possible declarative mechanisms for specification of more complex integrity constraints and their enforcement in the XML data model, is presented. As an example, the extended tuple constraint type is used.

4. The Extended Tuple Constraint in XML Data Model

In the XML Schema, three constraint types can be expressed as schema elements. The primary key constraint can be defined with the *xc:key* element in an XML Schema document. The foreign key constraint can be defined with the *xc:keyref* element in an XML Schema document. The unique constraint can be defined with the *xc:unique* element in an XML Schema document. Other constraint types that can be defined for the XML data model cannot be expressed in the current XML Schema specification. For that reason, in this paper we propose some extensions for the XML Schema specification that can enable the specification of more complex constraint types, like the extended tuple constraint.

In [1] we defined the extended tuple constraint (ETC) both in the relational and XML data model. There are no declarative mechanisms for the specification and enforcement of this type of constraints in the XML data model. For that reason we have decided to specify the ETC for the XML data model. In this section, we will repeat the definition of this constraint and in the Section 4 we will explain the code generator for the constraint validation in the XML DBMSs. The concrete example is the code generation for the validation of the extended tuple constraint, but this way of code generation is also applicable for other constraints we defined in our previous papers [2, 9]. It is also extendible since it can be done for every new defined constraint type.

The definition of the extended tuple constraint for the XML data model is similar to the definition of this constraint for the relational data model. ETC is defined between two or more element types. The mutual conditionality between the attribute values from different element types is specified by a logical condition. In [2] the constraint taxonomy and the constraint type formal specification for the XML data model are proposed. According to [2], the constraint type formal specification is a named 5-tuple. Therefore, the formal specification of the ETC type for the XML data model is given in the Definition 2.

Definition 2. Let E be an element type and $Attr(E)$ be a set of attributes specified within the element type E. The definition scope of an ETC in the XML model is an array of element types $E_1,..., E_m$. The logical condition is defined over a subset of the union of attribute sets $Attr(E_1)$, …, $Attr(E_m)$. The formal specification of an ETC is named 5-tuple:

$$\begin{aligned} &ExTupleCon(n,\\ &\quad t,\\ &\quad \tau_{ex}(E_1,..., E_m) = Constraint(E_1,..., E_m):Condition,\\ &\quad \tau_{ex}(E_1,..., E_m)(e) = Constraint(E_1,..., E_m)(e),\\ &\quad (e, \textit{unimportant role}, \{(insert,\{Restrict\}), (update, \{Restrict\})\})), \end{aligned}$$

where:
– *ExTupleCon* is the name (label) of the ETC type;
– Value n for the first tuple element specifies that more than one (at least two) element types have to be included in the specification of the ETC type;
– Value t for the second tuple element indicates the scope of the constraint type interpretation. For the ETC type it is a tuple constraint since it is interpreted over the values from only one tuple (element) gathered as the result of join operation between the instances (elements) of element types $E_1,..., E_m$;
– The third tuple element is the formula pattern $\tau_{ex}(E_1,..., E_m)=Constraint(E_1,..., E_m):Condition$, specifying that this constraint is defined over element types $E_1,..., E_m$, i.e. over the subset S of the union of attribute sets $Attr(E_1)$, … $Attr(E_m)$;
– The fourth tuple element indicates that constraint is interpreted over a tuple specified as e, that is gathered as the result of join operation between the instances (elements) of element types $E_1,..., E_m$ and whose set of attributes is the union of attribute sets $Attr(E_1)$, … $Attr(E_m)$; and
–The last tuple element is 3-tuple containing a tuple specified as e, role and set of pairs (*critical operation*, *set of actions*). For the ETC, role is not important, which means that each element type has the same role in the constraint. The critical operations for the ETC are insert or update of one of the instances (elements) of element types $E_1,..., E_m$. The critical operations may violate database consistency in regard to the ETC. For each critical operation, actions can be defined that would be carried out to maintain database consistency when this violation occurs. For both ETC type critical operations, the set of possible actions is a singleton that contains action *Restrict*. It means that if an ETC is violated the critical operation will not be executed. □

The extended tuple constraint is not supported in the XML Schema. As we have already mentioned, we propose the extension of the XML Schema with the element

xc:constraint where we can define which elements are involved in the ETC, as well as the additional condition that has to be satisfied. This extension is given in Fig. 2.

```
<xc:constraint
xmlns:xc="http://www.extendedconstraints.org/constraint"
type="xc:exTupleCon">
          <xc:condition from="<XPath>"
          fromName="<fNname>" to="<XPath>"
          toName="<tName>"
          toPK="<toPK>" keyref="attributes"
          additional="condition"/>
</xc:constraint>
```

Fig. 2. The extension of the XML schema for the extended tuple constraint type

The *xc:constraint* element has a child element *xc:condition*. The value *xc:exTupleCon* of the attribute *type* in element *xc:constraint* denotes that this is the extended tuple constraint. The *xc:condition* element in case of this constraint contains the following attributes: *from*, *to*, *fromName*, *toName*, *toPK*, *keyref* and *additional*. The values of these attributes are used in the process of the code generation.

The attributes *to* and *from* contain the XPath expressions used for selection of the nodes involved in the ETC check in the XML document. The *to* attribute denotes the XPath expression of the referencing (child) element and the *from* attribute denotes the XPath expression of the referenced (parent) element. The attributes *fromName* and *toName* contain names of the parent and child elements and are used for the message generation (when ETC is violated). The *toPK* attribute contains the name of the primary key in the child element. This attribute is used to identify the child element. The attribute *keyref* contains the name of the attribute which is the foreign key. The connection between the referenced and the referencing elements is described using the *keyref* attribute. The attribute *additional* contains the condition which is checked when connecting parent and child elements. This attribute is the core of the ETC check. It contains the condition which must be satisfied when the referenced and the referencing elements are paired.

Critical operations that can violate the extended tuple constraint are insert and update of a child element, denoted with the value of the *to* attribute, and update of a parent element, denoted with the value of the *from* attribute. If a child element is updated, that means that values of its non-key attributes or a value of its foreign key can be updated. If a parent element is updated, only values of the non-key attributes can be updated.

In order to generate the code for the validation of this, and in a similar way, other constraints, we need to describe in a formal way how these critical operations will be processed. We use a pseudo-code to explain what happens when a critical operation occurs. This pseudo-code will be used by the code generator to produce a code for validation of a certain constraint.

The pseudo-code for the validation of the extended tuple constraint when a new element, denoted with the value of the *to* attribute, is inserted, is given in Fig. 3. If a new referencing (child) element is inserted, first we have to check if there is a corresponding referenced (parent) element, denoted with the name of the *from* attribute. That is the referential integrity constraint. After that, an additional condition has to be checked and in this case this is the extended tuple constraint. If that condition is

satisfied, the insertion of the child element is enabled. This constraint is validated using XQuery function or trigger depending on the desired XML DBMS. The pseudo-codes for the validation of ETC in cases of update of parent or child element are given in [10].

```
input: e_to∈E_to
BEGIN insert_to
           IF(∃e_from∈E_from)(e_from@PK=e_to@FK)∧
         additional_condition=true
           THEN enable insert
           ELSE report error
END
```

Fig. 3. The pseudo-code for validation of ETC when a new element *to* is inserted

5. The Architecture of Code Generator for Constraint Validation

The main motivation for developing this code generator was to automate the process of the constraint validation. A constraint has to be specified with a pseudo-code, and then trough the code generator, using the extended XML Schema document, corresponding program code is generated. The generator produces two kinds of program code: XQuery functions and triggers. Two representative XML DBMSs are chosen: eXist [11] and Sedna [12]. eXist DBMS does not have a good support for triggers, so XQuery functions are generated for it. Sedna DBMS supports triggers completely and it can use generated triggers. The component diagram of the code generator is given in Fig. 4.

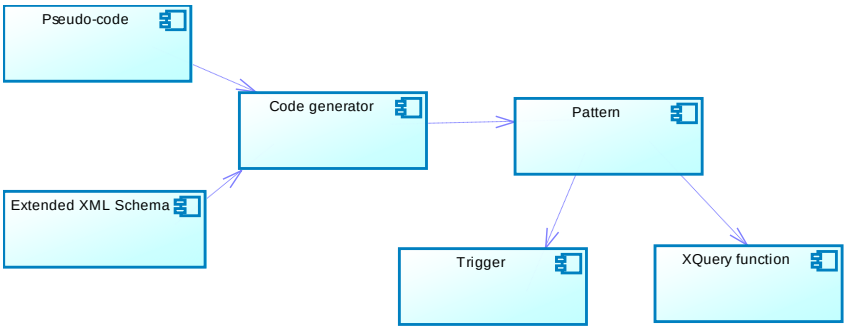

Fig. 4. The component diagram of the code generator

The main purpose of this generator is to automate the process of the constraint validation, using the proposed constraint notation. In that way, the code is generated instead of manual programming. The realization of the generator is based on the appropriate pseudo-code, such as the example of pseudo-code given in the Fig. 3. Other examples of pseudo-codes for validation of different constraint types and critical operations can be found in [10]. Based on the the pseudo-code, the appropriate templates are developed. During code generation, templates are filled according to the specification of the constraint type given in the extended XML Schema. In the case of ETC type it will be the extended XML Schema presented in Fig. 2. In sections 6 and 7

we demonstrate the process of the code generation for the ETC validation in the case of the target (child) element update.

The code generator is a Java application organized in modules. For each constraint type there is a specific package as well as a specific set of templates. In Fig. 5. there is a class diagram from the package for the generation of the code for validation of the extended tuple constraint type.

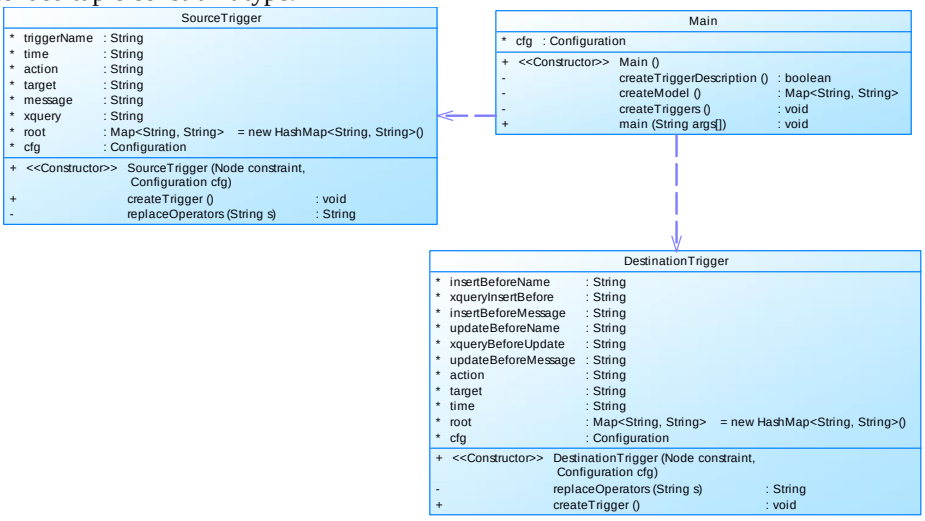

Fig. 5. The class diagram for the code generation for the validation of the extended tuple constraint

The main class in this package is the *Main* class. The method *createModel* reads the extended XML Schema, given in Fig. 2, and fills the template for generating of the description of the XQuery code. An example of such a description is given in Fig. 6. The method *createTriggers* reads the generated description of the XQuery code and according to the value of the attribute *type* of element *xc:trigger* (Fig. 6, lines 4, 12 and 20 of XQuery code description) instantiates the corresponding class from this package. If the value of the attribute *type* is *Source*, the object of the class *SourceTrigger* is created. If the value of the attribute type is *destination*, the object of the class *DestinationTrigger* is created. Both these two classes have the method *createTrigger*, which, according to the input parameters, fills the template and generates a trigger or an XQuery function.

The class *SourceTrigger* has methods that generate a trigger or a function, which are activated when a parent element is added or updated. The class *DestinationTrigger* generates a trigger or a function, which is activated when a child element is being processed. The described way of the code generation is extendible, because for each new constraint type a new package with its set of templates can be made.

6. The Code Generator for the Validation of the Extended Tuple Constraint

According to the pseudo-code with the description of the critical operations and the actions that have to be done, the code generator generates the description of the XQuery code. The description of the XQuery code is generated using the FreeMarker Java Template Engine library [13]. This library uses templates written in the FreeMarker notation. These templates are filled with the concrete data and according to them the output code is generated. The FreeMarker template for the XQuery code generation is given in Fig. 6.

```
<xc:triggers
xmlns:xc="http://www.extendedconstraints.org/constraint"
          type="xc:exTupleCon">
          <xc:trigger type="source"
name="ExTupleCon${fromName}" time="xc:before"
action="xc:update" target="${from}"
          xquery="($NEW/@${keyref}=$OLD/@${keyref}) and
count(${docName}${to}[@${keyref} = $NEW/@${keyref} and
not(${additionalSource})]) = 0" message="${fromName}
cannot be updated, it does not satisfy additional
condition" />
          <xc:trigger type="destination" name="
ExTupleCon${toName}BeforeInsert" action="xc:insert"
target="${to}" time="xc:before"
          xquery="exists(${docName}${from}[@${keyref} =
$NEW/@${keyref}]) and ${docName}${to}[@${keyref} =
$NEW/@${keyref}]/${additionalDestination})"
message="${toName} cannot be inserted, it does not
satisfy additional condition" />
          <xc:trigger type="destination"
name="ExTupleCon${toName}BeforeUpdate" action="xc:update"
target="${to}" time="xc:before"
           xquery="exists(${docName}${from}[@${keyref} =
$NEW/@${keyref}]) and ${docName}${to}[@${keyref} =
$NEW/@${keyref}]/${additionalDestination})"
message="${toName} cannot be updated, it does not satisfy
additional condition" />
</xc:triggers>
```

Fig. 6. The FreeMarker template for the XQuery code generation

The *xc:triggers* element can contain several *xc:trigger* elements. Each *xc:trigger* element describes the XQuery code that will be activated when the concrete action happens (insert, update, delete). The attributes *time* and *action* in the element *xc:trigger* define time and the type of the concrete action. The value of the attribute *time* can be *xc:before* or *xc:after*, which depends on the time of the trigger or function execution: before or after the concrete action. The value of the attribute *action* can be *xc:insert*, *xc:update* or *xc:delete*. The attribute *xquery* contains the XQuery code which will be

executed in the trigger or the function, and which will validate the given condition. The *message* attribute contains the text of the message that will be written to the user if the constraint is not satisfied. The *type* attribute determines which template will be used to create the trigger.

The trigger type can be *source* or *destination*. If the type is *source*, that trigger is generated when an action is done over the parent element. If the type is *destination*, that trigger is activated when an action is done over the child element.

Values in the expression ${*name*} denote the variables which will be replaced with the concrete values by the FreeMarker Template Engine. Those values are generated by the code generator, based on the XML Schema document.

The template given in Fig. 6 describes three critical operations that can violate the extended tuple constraint, as well as the time of the actions that are done to prevent the constraint violation. The first described action is taken before update of the parent element (Fig. 6, code line 6). The second and the third *xc:trigger* element contain the code generated for the insert or update of the child element (Fig. 6, code lines 13 and 21 respectively). Both actions are started before critical operation is done. The *xquery* attribute value contains the XQuery code generated according to the pseudo-code for certain critical operation. For example, the *xquery* attribute of the second *xc:trigger* element (Fig. 6, code lines 15–19) is generated based on pseudo-code presented in Fig. 3.

In this way, templates for all defined constraint types could be designed. From these templates, XQuery functions or triggers will be generated, which depends on the level of support for triggers in the selected XML DBMSs.

In Fig. 7. we present the template for the trigger that is generated based on the description given in the third *xc:trigger* element in Fig. 6, code lines 20–27. This trigger is executed before the update.

```
CREATE TRIGGER "${updateBeforeName}"
BEFORE REPLACE
ON ${collectionName}${target}
FOR EACH NODE
DO {
          if (${xqueryBeforeUpdate})
          then
               ($NEW)
          else
error(xs:QName("${updateBeforeName}"),"${updateBeforeMess
age}");}
```

Fig. 7. The template for before update trigger of the target (child) element

The variable ${*updateBeforeName*} contains the name of the trigger and it is filled from the *name* attribute in the *xc:trigger* element. The variable ${*collectionName*} is a global variable that contains the path to the collection in which the XML document is set. It is global because it does not exist in concrete nodes in the XML Schema documents and it cannot be read from them. The XPath expression to the node which is affected by the trigger, is in the variable ${*target*}. The variable ${*xqueryBeforeUpdate*} contains the XQuery code mentioned in the description of the XQuery code, i.e. in the *xquery* attribute in the *xc:trigger* element. Similarly, the

variable ${*updateBeforeMessage*} is filled based on the *message* attribute in the *xc:trigger* element.

If the corresponding XQuery function has to be generated based on the description given in the third *xc:trigger* element in Fig. 6, code lines 20–27, then the template given in Fig. 8 is used.

```
declare function local:canUpdate${toName}($OLD as
element(${toName}), $NEW as element(${toName}))
    as xs:boolean {
        let $ret := ${xquery}
        return $ret
};

declare function local:doUpdate${toName}($OLD as
element(${toName}), $NEW as element(${toName})) as
xs:boolean{
    let $i := update replace
${docName}${to}[@${toPK}=$OLD/@${toPK}] with $NEW
    return true()};

declare function local:update${toName}($OLD as
element(${toName}), $NEW as element(${toName})) as
xs:boolean{
    let $can :=  local:canUpdate${toName}($OLD, $NEW)
    let $res := if ($can)
                then local:doUpdate${toName}($OLD, $NEW)
                else false()
   return $res};
```

Fig. 8. The template for the XQuery function

The variable ${*docName*} is a global variable that contains the name of the document. The variable ${*toName*} contains the name of the element in the XML document, for which the XQuery function is declared and it is obtained from the XML Schema. The variable ${*xquery*} is filled from the element with the same name in the template (given in Fig. 6). The variable ${*to*} contains the XPath expression for selection of the node, which has to be updated. This value can be found in the extended XML Schema. The variable ${*toPK*} contains the name of the primary key from the parent element. Its value is also obtained from the extended XML Schema.

7. The Validation of the Extended Tuple Constraint by Generated Triggers and Functions

In the this section we describe what generated code looks like for the extended tuple constraint. The example corresponds with the Example 2, given in the Section 3. First we give the part of the XML Schema document extended with the *xc:constraint* element. The complete XML Schema document is given in [1].

The code generation for the extended tuple constraint validation starts with reading the XML Schema document. According to the constraint description given in the element *xc:constraint*, the description of the XQuery code, which is going to validate the constraint, is generated. In Fig. 9 the part of the XML Schema document, extended with the description of the condition for certain constraint, is given. This extension is used in the process of the code generation. This XML Schema document contains two element types: *Person* and *Document*. The primary key of the *Person* element is *PersId*. The primary key of the *Document* element is *DocNum*. There is a foreign key constraint in the *Document* element: *PersId* is the foreign key of the *Document* element. The additional condition is that the issue date of a document must be greater than or equal to the birth date of a person to whom that document belongs.

```
<xc:constraint type="xc:exTupleCon">
          <xc:condition from="/PDoc/Person"
fromName="Person" to="/PDoc/Document" toName="Document"
toPK="DocNum" keyref="PersId"
additional="${from}/@BirthDate ?lt; ${to}/@IssueDate"/>
</xc:constraint>
```

Fig. 9. The part of the XML Schema document with the extension for the extended tuple constraint

The *xc:constraint* element has an attribute *xc:exTupleCon* which means that it refers to ETC. As it is given in Fig. 2, the *xc:condition* element contains attributes: *from*, *to*, *fromName*, *toName*, *toPK*, *keyref* and *additional*. The *from* and *to* attributes contain the XPath expressions for the parent and child elements, respectively. In this example, these are */PDoc/Person* and */PDoc/Document*. The *fromName* and *toName* attributes have the names of the parent and child elements and in this example those are *Person* and *Document*. The *keyref* attribute contains the foreign key in the child element, and here it is *PersId*. The *toPK* attribute contains the name of the child element, and here it is *DocNum*. The *additional* attribute contains the condition that has to be satisfied: the *BirthDate* from the *Person* element has to be less than the *IssueDate* from the *Document* element. This condition connects the attributes from two element types – *Person* and *Document*, and this constraint is the extended tuple constraint.

Based on the XML Schema and the templates, the code generator generates the description of the XQuery code, which will be used either in trigger or in XQuery function. The description of the XQuery code is the XML document whose example is presented in Fig. 10. This document has all elements like the template given in Fig. 6, but all variables are now replaced with the concrete values by the FreeMarker Template Engine, i.e. names of the elements and attributes from the XML Schema document.

```
<xc:triggers
xmlns:xc="http://www.extendedconstraints.org/constraint"
type="xc:exTupleCon">
          <xc:trigger type="source"
name="ExTupleConPerson" time="xc:before"
action="xc:update" target="/PDoc/Person"
          xquery="($NEW/@PersId=$OLD/@PersId) and
count(doc('PDoc.xml')/PDoc/Document[@PersId =
$NEW/@PersId and not($NEW/@BirthDate ?lt; @IssueDate)]) =
```

```
0" message="Person cannot be updated, it does not satisfy
additional condition" />
          <xc:trigger type="destination"
name="ExTupleConDocumentBeforeInsert"
          action="xc:insert" target="/PDoc/Document"
time="xc:before"
          xquery="exists(doc('PDoc.xml')/PDoc/Person[@Per
sId = $NEW/@PersId]) and
doc('PDoc.xml')/PDoc/Document[@PDoc =
$NEW/@PDoc]/@BirthDate ?lt; $NEW/@IssueDate)"
message="Document cannot be inserted, it does not satisfy
additional condition" />
          <xc:trigger type="destination"
name="ExTupleConDocumentBeforeUpdate" action="xc:update"
target="/PDoc/Document" time="xc:before"
          xquery="exists(doc('PDoc.xml')/PDoc/Person[@Per
sId = $NEW/@PersId]) and
doc('PDoc.xml')/PDoc/Document[@PersId =
$NEW/@PersId]/@BirthDate ?lt; $NEW/@IssueDate)"
message="Document cannot be updated, it does not satisfy
additional condition" />
</xc:triggers>
```

Fig. 10. The description of the XQuery code for realization of the extended tuple constraint

Based on this document either trigger or the XQuery function will be generated using FreeMarker Template Engine library. The trigger generated according to the template that is specified in the third *xc:trigger* element in Fig. 10, is given in Fig. 11.

```
CREATE TRIGGER "ExTupleConDocumentBeforeUpdate"
BEFORE REPLACE
ON collection('ExTupleCon')/PDoc/Document
FOR EACH NODE
DO {
          if (exists(fn:doc('PDoc',
'ExTupleCon')/PDoc/Person[@PersId = $NEW/@PersId]) and
fn:doc('PDoc', 'ExTupleCon')/PDoc/Document[@PersId =
$NEW/@PersId]/@BirthDate < $NEW/@IssueDate))
          then
            ($NEW)
          else
          error(xs:QName("ExTupleConDocumentBeforeUpdate"
),"  Document cannot be updated, it does not satisfy
additional condition");}
```

Fig. 11. The trigger generated according to the described template in Fig. 10

The XQuery function generated according to the template that is specified in the third *xc:trigger* element in Fig. 10 is given in Fig. 12.

```
declare function local:canUpdateDocument($OLD as
element(Document), $NEW as element(Document)) as
xs:boolean {
        let $ret :=
exists(doc('PDoc.xml')/PDoc/Person[@PersId =
$NEW/@PersId]) and doc('PDoc.xml')/PDoc/Document[@PersId
= $NEW/@PersId]/@BirthDate < $NEW/@IssueDate)
        return $ret
};

declare function local:doUpdateDocument($OLD as
element(Document), $NEW as element(Document)) as
xs:boolean{
        let $i := update replace
doc('PDoc.xml')/PDoc/Document[@DocNum=$OLD/@DocNum] with
$NEW
        return true()};

declare function local:updateDocument($OLD as
element(Document), $NEW as element(Document)) as
xs:boolean{
        let $can :=  local:canUpdateDocument($OLD, $NEW)
        let $res := if ($can)
                    then local:doUpdateDocument($OLD,
$NEW)
                    else false()
        return $res};
```

Fig. 12. The XQuery function generated according to the described template in Fig. 10

In this example we have demonstrated the process of the code generation. This is only the part of the generated code that refers to the validation of the extended tuple constraint when the element is updated. The main purpose of this generator is to automate the process of the constraint validation, using the presented notation. That way, instead of manual programming, the code for of trigger and XQuery functions for constraint validation is generated. The same principle can be used for validation of any type of constraint.

8. Related Work

The extended tuple constraint is one of the constraints that is used in practice but there are few papers that define and explain this type of constraint. In [8] authors give an example of the relational database schema where one of the constraints is the extended tuple constraint. A common classification scheme for data integrity constraints in relational data model, according to the scope of data that is being constrained, comprises of: attribute, tuple, table and database constraints. ETC can be classified as a database constraint. The SQL2 standard [14] has introduced the ASSERTION construct as the most general means to express arbitrary integrity constraints declaratively in

SQL. Assertions are defined on database schema level and therefore can be used to declare complex constraints, including ETCs. Most current RDBMSs do not support CREATE ASSERTION at all. Even if the support exists, like in the Ocelot DBMS [15], it is limited. Koppelaars explains in [16] why it is hard to develop the algorithms that parse an assertion and then figures out when and how to most efficiently validate the assertion. There are alternative solutions to simulate assertions using some mechanisms available depending on the RDBMS used. Using of updateable views with the check option to simulate assertions is limited on cases where the assertion only depends on updates in one table. ETCs are constraints that tackle at least two (not necessarily different) tables and therefore mechanisms based on the updateable views are not sufficient.

Although DBMSs have offered a few declarative mechanisms to implement constraints, they are still limited on built-in constraints. For the vast majority of constraints database designers and programmers have to develop their own mechanisms to implement them. One of the main problems in the context of ETCs is the scope of data that is being constrained. For a DBMS vendor it is hard to come up with an efficient ETC's implementation. The algorithm would be developed that parses an ETC specification and then figures out the events that will trigger the action of constraint validation and how to most efficiently validate the ECT. According to Koppelaars DBMSs vendors have not been able to develop such an algorithm for an arbitrary assertion, yet. It is one of the reasons for the lack of the literature that tackles more complex constraint types in general, and ETCs in particular. RuleGen [17] is an example of a framework, written in PL/SQL that aids in implementing data integrity constraints inside the Oracle RDBMS. It generates PL/SQL code to maintain some kinds of database type of data integrity constraints that could be applied for ETCs implementation, too.

In current XML specifications, XML Schema definitions include type definitions, occurrence cardinalities, unique constraints, and referential integrity. There are multiple papers dealing with these types of XML constraints [18–22]. A generic constraint definition language for XML, with expressive power comparable to aforementioned assertions in relational DBMSs, is still not present in the XML Schema specification. To the best of our knowledge there are no papers on more complex constraint types, like the extended tuple constraint in the XML data model.

XML/XQuery code generator [23], provides the synthetic XML data generation. It has rich support for data types and generates complex XQueries that are compatible with generated XML data. This generator does not have any support for generating code for the constraint validation of that XML data. In [24] generation of the complex XML content is available, and the tool allows the specification of most common integrity constraints over the data in lists, like ID, IDREF, uniqueness. Integrity constraints over element or attributes can also be specified, which allows the generation of consistent documents. There is no support for more complex constraints in the XML document.

In [25], an approach for defining inclusion dependencies for XML in XML Schema, which is based on paths, is proposed. Two existing components, selector and field, are used to locate the information items, which use restricted XPath as a path description language. The semantics of the field components to support a set (list) of nodes and node with complex type is extended. Dealing with null values and comparing element nodes with complex type are also discussed.

XML documents can have relational or hierarchical structure [26]. In [2] we have discussed these two structures. We have concluded that the relational structure is a better way of creating XML documents if we want to specify different types of constraints. Documents created in relational way avoid redundancy, but they are large. If the hierarchical structure is used, documents are more readable, but it is impossible to specify and implement constraints in the way we used to do in [2, 9]. That is the reason why we have adopted the relational way of modeling XML documents. It means that all elements are on the same level under the root element.

The conclusions presented in [16] can be applied in the context of the constraint implementation in XML DBMSs. The usual way of implementing constraints in XML DBMSs is using triggers, as stated in [27]. The XML DBMS that supports triggers in a way similar to relational databases is Sedna [12]. On the other hand, there are XML DBMSs which do not support triggers in the appropriate extent, such as eXist [11] and for those XML DBMSs, we implement constraints by means of XQuery functions [28]. In this paper we present the code generator that can generate both triggers and XQuery functions.

9. Conclusion

Integrity constraints have always been an important part of the database design and implementation. Its importance grows with increasing demands regarding the quality and reliability of data. Integrity constraint specifications are translated into constraint enforcing mechanisms provided by a DBMS used to implement a database. Most of the commercial DBMSs offer efficient declarative support for the well-known constraint types like domain constraints, attribute and tuple constraints, uniqueness constraints and foreign key constraints. More complex constraint types, like the extended tuple constraint, are mostly disregarded by actual RDBMSs and XML DBMSs forcing the users to manage them via custom procedures or triggers. That is the reason why these types of constraints are ignored by database designers in a way that they do not recognize, specify and implement them.

In this paper we have presented the classification of the constraint types in the relational data model, discussed possible declarative mechanisms for their specification and enforcement in the XML data model, and illustrated our approach to the definition and implementation of complex constraint types in the XML data model on the example of the extended tuple constraint type. We have also proposed one approach to the generation of program code for enforcing constraints in XML DBMSs. In order to avoid a lot of manual programming we have proposed a code generator which can generate code for any specified constraint type. The extension of the XML Schema must be defined for a certain constraint type, as well as a pseudo-code with the critical operations and actions undertaken in the case of violation of that constraint type. The generated code can be applied to any constraint of that constraint type in the XML document. We have also given the example of usage of this code generator for enforcing the extended tuple constraint.

Our approach offers control of constraints using XQuery functions and triggers, depending on the level of support for triggers in XML DBMSs. The usual way of implementing constraints in XML DBMSs is using triggers, but not all XML DBMSs

have support for triggers in the appropriate extent. We have chosen eXist DBMS to implement constraints using XQuery functions, and Sedna DBMS to work with triggers.

Further development is directed towards comparing presented implementations and getting experimental results based on the benchmark data.

During and after the development of the presented code generator, we have tested it on various examples. The main drawback of the evaluation by example is that it is performed by the authors of the generator and therefore may be biased. The objective qualitative evaluation of generated artifacts for both ETCs presented in this paper and ERICs presented in [3] would be conducted. The number of code lines in handmade code vs. number of code lines in generated code could be informative, but not substantial. Therefore, we plan to perform the evaluation of the presented code generator with the evaluation participants with different levels of knowledge and skills concerning database design and programming. Characteristics that would be evaluated, beyond others, are functional suitability, usability, trustworthiness, expressiveness, and productivity.

Acknowledgments

The research presented in this paper was supported by the Ministry of Education, Science and Technological Development of Republic of Serbia, Grant OI-174023.

10. References

1. Vidaković, J., Ristić, S., Kordić, S., Luković, I.: Extended Tuple Constraint Type in Relational and XML Data Model – Definition and Enforcement. In Proceedings of BCI '17, September 20–23, 2017, Skopje, Macedonia, 8 pages. https://doi.org/10.1145/3136273.3136294
2. Vidaković, J., Luković, I., Kordić, S.: Specification and Implementation of the Inverse Referential Integrity Constraint in XML Databases, 7th Balkan Conference in Informatics, 2015, ACM New York, USA, ISBN 978-1-4503-33351, DOI: 10.1145/2801081.2801111
3. Vidaković, J., Ristić, S., Kordić, S., Luković, I., The Extended Referential Integrity Constraint Type – Specification and Implementation in Relational and XML Database Management Systems, Proceedings of the 8th International Conference on Information Society and Technology, 2018, Kopaonik, Serbia, Vol.1, pp.143-148
4. Date, C. J., Darwen, H.: Types and the Relational Model. The Third Manifesto, 3rd ed. Addison Wesley, 2006.
5. Eessaar, E.: "Using Meta-modeling in order to Evaluate Data Models", In Proceedings of the 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases, Corfu Island, Greece, February 16-19, 2007.
6. Elmasri, R., Navathe, B.S.: Database Systems: Models, Languages, Design and Application Programming, Sixth Edition, Pearson Global Edition, ISBN 978-0-13-214498-8. 2011.
7. Ristić, S., Aleksić, S., Čeliković, M., Luković, I.: Generic and Standard Database Constraint Meta-Models, Computer Science and Information Systems 11(2):679–696, June 2014, doi:10.2298/CSIS140216037R
8. Ristić, S., Aleksić, S., Čeliković, M., Luković, I.: An EMF Ecore based Relational DB Schema Meta-Model, 6. International Conference on Information Technology - ICIT, Amman: AL-Zaytoonah University of Jordan, 8-10 May, 2013, pp. 1-12, ISBN 978-9957-8583-1

9. Vidaković, J., Luković, I., Kordić, S., Specification and Validation of the Referential Integrity Constraint in XML Databases, Proceedings of the 6th International Conference on Information Society and Technology, 2016, Kopaonik, Serbia, ISBN 978-86-85525-18-6, pp. 197-202.
10. Vidaković, J.: Specification and Validation of Constraints in XML Data Model, Ph.D. thesis, University of Novi Sad, Faculty of Technical Sciences, 2014. (In Serbian)
11. eXist, http://exist-db.org/exist/apps/homepage/index.html (July 2018)
12. Sedna, Native XML Database System, www.sedna.org (July 2018)
13. FreeMarker Java Template Engine, http://freemarker.org/ (July 2018)
14. SQL92 (SQL2) ISO (International Standards Organization) 9075: 1992, Database Language SQL; ANSI (American National Standards Institute) X3.135-1992, Database Language SQL
15. Ocelot: OCELOTSQL Homepage (2001). On-line, available at: https://ocelot.ca/dbms.htm
16. Koppelaars, T.: CREATE ASSERTION: The Impossible Dream?, NoCOUG Journal, Vol. 27, No.3, Avgust 2013, pp. 13–15. 2013.
17. RuleGen, data quality assurance, https://github.com/Mrvek/RuleGen (July 2018)
18. Fan, W.: XML constraints: Specification, analysis, and applications. In H. Christiansen, & D. Martinenghi (Eds.), LAAIC'05, Proceedings of the 1stInternational Workshop on Logical Aspects and Applications of Integrity Constraints. Included in Proceedings of DEXA 2005, International Workshop on Database and Expert Systems Applications, Copenhagen, Denmark, pp. 805-809, 2005.
19. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.: Reasoning about keys for XML. Information Systems, 28(8), pp. 1037-1063, 2003.
20. Buneman, P., Fan, W., Weinstein, S.: Interaction between path and type constraints. ACM Trans. Comput. Log., 4(4), pp.530-577, 2003.
21. Alon N., Milo, T., Neven, F. , Suciu, D., Vianu V.: XML with data values: Typechecking revisited. J. Comput. Syst. Sci., 66(4), 688-727, 2003.
22. Klarlund, N., Schwentick, T., Suciu, D.: XML: Model, schemas, types, logics, and queries. In J. Chomicki, R. van der Meyden, & G. Saake (Eds.), Logics for Emerging Applications of Databases [outcome of a Dagstuhl seminar]. Springer. 2003.
23. Todić, M., Uzelac, B.: Combined XML/XQuery generator, DBTest'12 Proceedings of the Fifth International Workshop on Testing Database Systems, Scottsdale, Arizona, ACM, May 2012, doi: 10.1145/2304510.2304519
24. Barbosa, D., Mendelzon, A., Keenleyside, J., Lyons, K.:ToXgene: a template-based data generator for XML, Proceedings of the 2002 ACM SIGMOD international conference on Management of Data, Madison, Wisconsin, doi: 10.1145/564691.564769
25. Chen, H., Liao, H., Integrity constraints for XML, Proceedings of the 2010 IEEE International Conference on Software Engineering and Service Sciences, Beijing, 2010, pp. 331-334, doi: 10.1109/ICSESS.2010.5552445
26. Chaudhuri, N., Glace, J., Wilson, G.: Hierarchical vs. Relational XML Schema Designs, A Study for the environmental Council of States, Report ECO41T1, http://www.exchangenetwork.net/dev_schema/schemadesigntype.pdf, June 2006.
27. Grinev, M., Rekouts, M., Introducing Trigger Support to XML Database Systems, Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems SYRCoDIS, St. Petersburg, Russia, 2005.
28. XQuery, http://www.w3.org/TR/xquery/ (July 2018)

Jovana Vidaković received her bachelor and Mr (2 years) degrees from the Faculty of Sciences, University of Novi Sad, and Ph.D. degree from Faculty of Technical Sciences, University of Novi Sad. Currently, she works as an Assistant Professor at the Faculty of Sciences, University of Novi Sad, where she lectures in several Computer Sciensec and Informatics courses. Her research interests are related to Database Systems and Information Systems.

Sonja Ristić works as a full professor at the University of Novi Sad, Faculty of Technical Sciences, Serbia. She received two bachelor degrees with honors from UNS, one in Mathematics, Faculty of Science in 1983, and the other in Economics from Faculty of Economics, in 1989. She received her Mr (2 year) and Ph.D. degrees in Informatics, both from Faculty of Economics (UNS), in 1994 and 2003. From 1984 till 1990 she worked with the Novi Sad Cable Company NOVKABEL–Factory of Electronic Computers. From 1990 till 2006 she was with High School of Business Studies–Novi Sad, and since 2006 she has been with the FTS (UNS). Her research interests are related to Database Systems and Software Engineering. She is the author or co-author of over 70 papers, and 10 industry projects and software solutions in the area.

Slavica Kordić received her M.Sc. degree from the Faculty of Technical Sciences, at University of Novi Sad. She completed her Mr (2 year) and Ph.D. degrees, both from Faculty Technical Sciences, University of Novi Sad. Currently, she works as an Assistant Professor at the Faculty of Technical Sciences at the University of Novi Sad, where she lectures in several Computer Science and Informatics courses. Her research interests are related to Information Systems, Database Systems and Model Driven Software Engineering.

Ivan Luković received his M.Sc. 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, Business Intelligence Systems and Software Engineering. He is the author or coauthor of over 150 papers, 4 books, and 30 industry projects and software solutions in the area.

Received: March 24, 2018; Accepted: September 3, 2018