The Structure of A Subschema and Its XML Specification

Ivan Luković
University of Novi Sad, Yugoslavia
Faculty of Technical Sciences, Novi Sad
ivan@iis.ns.ac.yu

Pavle Mogin
Victoria University of Wellington, New Zeland
School of Mathematical and Computing Sciences, Wellington
pmogin@mcs.vuw.ac.nz

Miro Govedarica
University of Novi Sad, Yugoslavia
Faculty of Technical Sciences, Novi Sad
miro@iis.ns.ac.yu

Sonja Ristić
Business Collage, Novi Sad, Yugoslavia
sdristic@uns.ns.ac.yu

Abstract: *The notion of a* subschema*, as a formal and abstract definition of data, constraints, and database update activities that are needed to make a transaction program, is introduced in the paper. A subschema is designed with respect to a user requirement and an existing relational database schema. A subschema design is done in the course of a transaction program design. Establishing methods for the design of a transaction program specification asks for a definition of a general and formal structure of that program. Our recent research shows that this goal may be achieved by means of XML specifications. Using XML specifications provides an automatic procedure for generating an executable specification of a transaction program. In the paper we present a formal way to describe the structure of a subschema by means of XML. This description supports SQL subschema specification using concepts of ANSI SQL92 standard.*

Keywords: *Relational Database Schema, Transaction Program Specification, Subschema, XML Specification.*

1. INTRODUCTION

The software component of an information system (IS) can be observed as a set of applications. Each application is a structured set of transaction programs, which is usually aimed at supporting logically related processes that are performed in the same business unit. Specifying transaction programs and applications is an important and time-consuming methodological task in the process of the design of an IS. The design specification of a transaction program is a formalized description of that program. It is aimed at supporting the implementation of an end user business task that is defined by means of a user request. It is usually assumed that the transaction programs will be executed against a database. Accordingly, the basic structural components of a transaction program specification are: (i) A specification of the human-computer interface; (ii) A data definition; and (iii) A formal description of a data processing procedure. In this paper, the data definition part of the transaction program specification is called a *subschema*. A subschema is a formal and abstract definition of data, constraints, and database update activities that are needed to make a transaction program.

Establishing methods for the design of a transaction program specification and an application specification asks for a definition of a general and formal structure of those specifications. It is particularly important if an application development is based on using a code generator. In order to make a soft-

ware development process satisfactory effective, nowadays it is practically inevitable to use an integrated CASE tool with the appropriate code generators included. At the other hand, a requirement to design a program and application specifications that are fully independent of a chosen programming and run-time environment becomes highly important so as to preserve investments in a software development for a longer time.

Our recent research shows that this goal may be achieved by means of XML [9] and UIML [8] specifications. It appears that XML and UIML are suitable languages for specifying a general structure of a program specification and an application specification. Using XML and UIML provides an automatic procedure for generating executable specifications of transaction programs and applications in a chosen programming and run-time environment. A process of generating an executable XML program and application specification is shown in Figure 1. The process input elements are: (i) the repository information of a chosen CASE tool; (ii) UIML DTD Specification [8]; (iii) a common specification of the application user interface designed using UIML [3]; and (iv) DTDs as formal representations of the subschema and the application structure. It is supposed that the repository supports all the concepts necessary to design a program and an application specification. The process includes two techniques: (i) reading the repository information and an automatic producing of the corresponding XML/UIML code and (ii) merging of XML documents by applying XSL transformations [10]. More details about XML and UIML usage for these purposes may be found in [3].

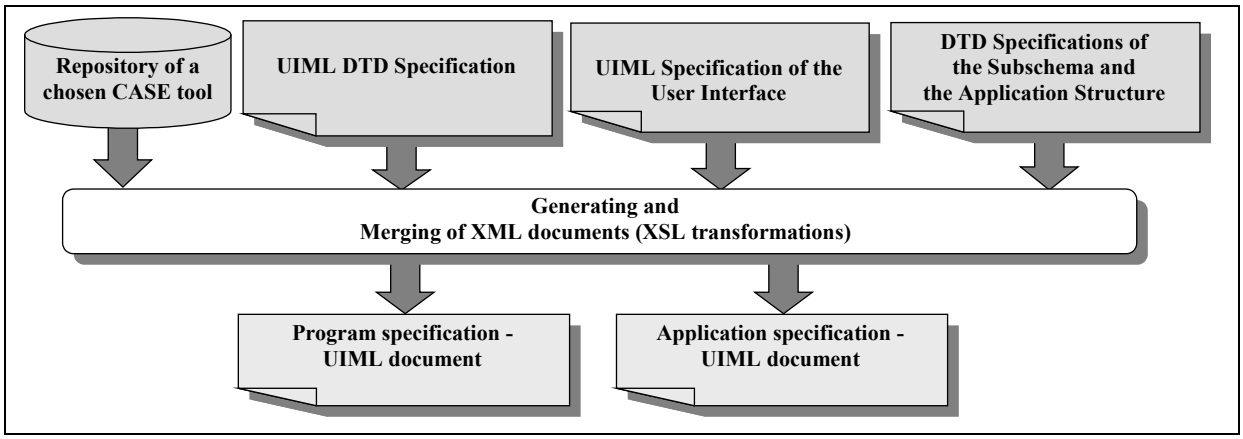

Figure 1. A Concept of Generating Executable Program Specifications

This paper introduces the notion of a subschema and presents a way of formal specification of a subschema by means of XML. Beside the Introduction and the Conclusion, the paper contains three additional sections. Section 2 presents a formalization of commonly used integrity constraint types, necessary to build both a database schema and a subschema. A concept of subschema is introduced in Section 3. In Section 4, an XML DTD specification of a subschema general structure is presented.

2. PRELIMINARIES

A relational database schema is a pair $(S, \mathcal{I})$, where S is a set of relation schemes and $\mathcal{I}$ is a set of interrelation constraints. It is supposed in the paper that the database schema is produced using a well-defined methodological approach, and no further attention is paid to this issue.

Each relation scheme from S is a named triple: $N(R, C, K_p(R))$, where N is a unique name, R is an attribute set, and C is a specification of constraints. A relation scheme will be often referred simply by its name N. The specification of constraints C is a triple $(\mathcal{K}, \tau(N), \mathcal{U}niq(N))$, where $\mathcal{K}$ is a set of *keys*, $\tau(N)$ will be called *tuple integrity constraint*, and $\mathcal{U}niq(N)$ is a (possible empty) set of *uniqueness constraints*. Each key $X \in \mathcal{K}$ is a set of non nullable attributes from R and each value of X uniquely identifies at most one tuple from an instance over N and there is no proper subset of X with the same property. $K_p(R) \in \mathcal{K}$ denotes the *primary key* of the relation scheme N. The definitions of the other two integrity constraint types are given in the separate sections.

The set of interrelation constraints $\mathcal{I}$ may contain various types of constraints, of which frequently used referential integrity constraint is just one.

In order to define the tuple integrity constraint, let us first introduce notions of: (i) a domain integrity constraint, (ii) a domain integrity constraint of an attribute in a universal set and (iii) a domain integrity constraint of an attribute in a relation scheme.

2.1. THE DOMAIN INTEGRITY CONSTRAINT

Informally, a domain of an attribute A is a set of such values that can be assigned to A. This set can be specified by defining a *data type*, a *maximal data length* and a *condition* that each constant, which is supposed to be a domain value, has to satisfy. There are two classes of domains: *predefined* (*primitive*) and *user defined*. Let $\mathcal{OP}$ denote a set of predefined domains. It is represented by means of implicitly defined, initially existing data types in a database management system (DBMS), for example, $\mathcal{OP}$ = {*Character, Integer, Real, Logical, Date,...*}. User defined domain is explicitly specified by means of a previously specified domain, using the principle of inheritance.

Definition 1. Let $\mathcal{O}$ be a set of all the domains in a system, such that $\mathcal{OP} \subseteq \mathcal{O}$ holds. Let D be a domain from $\mathcal{O}$. A *domain integrity constraint* $\mathcal{id}(D)$ is a triple:

$$\mathcal{id}(D) = (\mathcal{Typ}(D), \mathcal{Len}(D), \mathcal{Con}(D)), \qquad (1)$$

where $\mathcal{Typ}(D)$ is a *super domain* (*type constraint*), $\mathcal{Len}(D)$ is a *maximal length constraint* and $\mathcal{Con}(D)$ is a *logical condition*. If any component in (1) is not specified for a domain $D \in \mathcal{O}$, it will be denoted with an "empty" symbol Δ. □

Domain constraint for a predefined domain $D_p \in \mathcal{OP}$ is not explicitly specified. Thus, $\mathcal{Typ}(D_p) = \Delta$, and $\mathcal{Con}(D_p) = \Delta$ holds, for each $D_p \in \mathcal{OP}$. It is supposed that there are some predefined domains in a DBMS for which $\mathcal{Len}(D_p)$ has to be specified (for example *Character*). Otherwise, $\mathcal{Len}(D_p) = \Delta$ holds.

A specification of the type constraint $\mathcal{Typ}$ defines a relation of *domain inheritance* $\mathcal{Typ} \subseteq \mathcal{O}^2$ in a following way. $(D_i, D_j) \in \mathcal{Typ}$ iff $D_j = \mathcal{Typ}(D_i)$. $\mathcal{O}$ can be considered as *well-defined set* of domains, if the graph $(\mathcal{O}, \mathcal{Typ})$ is an acyclic structure such that:

$$(\forall D_i \in \mathcal{O})(\neg(\exists D_j \in \mathcal{O})((D_i, D_j) \in \mathcal{Typ}) \Rightarrow D_i \in \mathcal{OP})) \qquad (2)$$

holds. In other words, if $\mathcal{O}$ is a well-defined set, then type constraint of a new domain is always specified by using a previously specified domain from $\mathcal{O}$. For each domain, after a finite number of times the relation $\mathcal{Typ}$ is applied, a predefined domain must be reached. In this paper, we suppose that $\mathcal{O}$ is a well-defined set of domains.

For each domain $D_i \in \mathcal{O} \setminus \mathcal{OP}$, $\mathcal{Len}(D_i)$ is specified according the following rules. $\mathcal{Len}(D_i) = \Delta$, iff $\mathcal{Typ}(D_i) \in \mathcal{O} \setminus \mathcal{OP}$, or $\mathcal{Typ}(D_i) \in \mathcal{OP}$ and $\mathcal{Len}(\mathcal{Typ}(D_i)) = \Delta$. Otherwise, $\mathcal{Len}(D_i) \leq \mathcal{Len}(\mathcal{Typ}(D_i))$, iff $\mathcal{Typ}(D_i) \in \mathcal{OP}$ and $\mathcal{Len}(\mathcal{Typ}(D_i)) \neq \Delta$.

A logical condition $\mathcal{Con}(D)$ is composed of atomic conditions, related by logical operators $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$. Each atomic condition is of the form $d\ \theta\ k$, $k\ \theta\ d$, or $d \in \{k_1,..., k_n\}$, where d is a formal variable, representing a value, $k_1,..., k_n$ and k are the constants and $\theta \in \{<, >, \leq, \geq, \neq, =\}$. Complete definitions of an atomic condition and a condition $\mathcal{Con}(D)$ can be found in [5] and [7].

A domain constraint $\mathcal{id}(D)$ may be validated for any value d_i. A validation is denoted as $\mathcal{id}(D)(d_i)$, and $\mathcal{id}(D)(d_i) \in \{\mathsf{T}, \perp\}$ must hold, where T stands for *true*, and $\perp$ stands for *false*. In the following text, validation rules for $\mathcal{id}(D)(d_i)$ are presented.

- If d_i is a *null value* (*missing value*), which is denoted by $d_i = \omega$, then $\mathcal{id}(D)(d_i) = \mathsf{T}$. Thus, a null value satisfies each domain constraint by default.
- Validation rules for each $D \in \mathcal{OP}$ are embedded in a DBMS and no further attention is paid to them.

- If $D \in \mathcal{O} \setminus \mathcal{O}p$, then a validation $\mathcal{id}(D)(d_i)$ is defined by the expression:

$$\mathcal{id}(D)(d_i) = \mathcal{id}(\mathcal{T}yp(D))(d_i) \wedge \mathcal{L}en(D)(d_i) \wedge \mathcal{C}on(D)(d_i). \quad (3)$$

  - A validation $\mathcal{id}(\mathcal{T}yp(D))(d_i)$ is performed using the same rules on a domain specified by $\mathcal{T}yp(D)$ in a recursive manner.
  - $\mathcal{L}en(D)(d_i)$ is validated only if $\mathcal{id}(\mathcal{T}yp(D))(d_i) = \mathsf{T}$ holds. If $\mathcal{L}en(D) = \Delta$, then $\mathcal{L}en(D)(d_i) = \mathsf{T}$. If $\mathcal{L}en(D) \neq \Delta$, then $\mathcal{L}en(D)(d_i) = \mathsf{T}$ iff the length of d_i is not grater than the specified $\mathcal{L}en(D)$.
  - $\mathcal{C}on(D)(d_i)$ is validated only if $\mathcal{L}en(D)(d_i) = \mathsf{T}$ holds. Besides, it is supposed that $\mathcal{C}on(D)$ is built in such a way that all the constants from $\mathcal{C}on(D)$ satisfy both $\mathcal{T}yp(D)$ and $\mathcal{L}en(D)$. If $\mathcal{C}on(D) = \Delta$, then $\mathcal{C}on(D)(d_i) = \mathsf{T}$. If $\mathcal{C}on(D) \neq \Delta$, then $\mathcal{C}on(D)(d_i) = \mathsf{T}$ iff d_i satisfies the logical condition $\mathcal{C}on(D)$, which is transformed in such a way that d_i replaces all the occurrences of a formal variable d. All the relational and logical operators are validated in a usual way.

The relation $\mathcal{T}yp \subseteq \mathcal{O}^2$ allows defining a notion of the domain chain. Let $\mathcal{L}\mathcal{d}(D) = (D_n, ..., D_1)$, $n \geq 2$, be a sequence of domains, where $D, D_1, ..., D_n \in \mathcal{O}$ and $D \notin \mathcal{O}p$. $\mathcal{L}\mathcal{d}(D)$ is called a *domain chain* for D, if $D_1 \in \mathcal{O}p$, $D_n = D$, and $(\forall i \in \{2, ..., n\})(\mathcal{T}yp(D_i) = D_{i-1})$ holds. The following lemma presents a rule how to validate the domain constraint of a user-defined domain. The proof may be found in [7].

Lemma 1. Let D be a user defined domain and let $\mathcal{L}\mathcal{d}(D) = (D_n, ..., D_1)$ be a domain chain for D. Then

$$\mathcal{id}(D)(d_p) = \mathcal{id}(D_1)(d_p) \wedge \mathcal{L}en(D_2)(d_p) \wedge \left(\bigwedge_{i=2}^{n} \mathcal{C}on(D_i)(d_p) \right) \quad (4)$$

holds. □

Example 1. Let a predefined domain $Integer \in \mathcal{O}p$ be given. Let domains *DMarkGrade* and *DPositiveMarkGrade* be defined in the following way: $\mathcal{id}(DMarkGrade) = (Integer, 2, d \geq 5 \wedge d \leq 10)$, and $\mathcal{id}(DPositiveMarkGrade) = (DMarkGrade, \Delta, d \geq 6)$. A domain chain for *DPositiveMarkGrade* is (*DPositiveMarkGrade*, *DMarkGrade*, *Integer*). According to Lemma 1, $\mathcal{id}(DPositiveMarkGrade)(d_p)$ will be validated using the formulae:

$$\mathcal{id}(Integer)(d_p) \wedge \mathcal{L}en(DMarkGrade)(d_p) \wedge \mathcal{C}on(DMarkGrade) \wedge \mathcal{C}on(DPositiveMarkGrade) =$$
$$\mathcal{id}(Integer)(d_p) \wedge (2)(d_p) \wedge (d \geq 5 \wedge d \leq 10)(d_p) \wedge (d \geq 6)(d_p). \square$$

2.2. THE DOMAIN INTEGRITY CONSTRAINT OF AN ATTRIBUTE IN A UNIVERSAL SET

Let a universal relation scheme $(\mathcal{U}, C)$ be given, where $\mathcal{U} = \{A_i \mid i \in \{1, ..., n\}\}$ is a universal set of attributes and C is a set of global constraints. Let $\mathcal{O}$ be a set of all domains and let $\mathcal{id}(\mathcal{O}) = \{\mathcal{id}(D) \mid D \in \mathcal{O}\}$ denote a set of all domain constraints, where $\mathcal{id}(\mathcal{O}) \subseteq C$ holds. Each constraint from $\mathcal{id}(\mathcal{O})$ is specified independently of any attribute from $\mathcal{U}$.

A domain function $Dom_{\mathcal{U}}: \mathcal{U} \rightarrow \mathcal{O}$ associates each attribute $A \in \mathcal{U}$ with a domain from $\mathcal{O}$. Consequently, $Dom_{\mathcal{U}}$ associates the corresponding domain constraint $\mathcal{id}(D)$ with A. Let $\mathcal{D}om(\mathcal{U}, A)$ denote a domain D for which $Dom_{\mathcal{U}}(A, D)$ holds. Then a domain constraint $\mathcal{id}(\mathcal{D}om(\mathcal{U}, A))$ will be called the *domain constraint of an attribute in a universal set of attributes*. $\mathcal{id}(\mathcal{D}om(\mathcal{U}, A))$ may be validated for an attribute A value for each tuple $t \in Tuple(\mathcal{U})$, which is denoted as $\mathcal{id}(\mathcal{D}om(\mathcal{U}, A))(t[A])$. Validation is performed according to rules for $\mathcal{id}(D)$, presented in Lemma 1.

2.3. THE DOMAIN INTEGRITY CONSTRAINT OF AN ATTRIBUTE IN A RELATION SCHEME

We suppose that a database schema $(S, \mathcal{I})$ is designed by a decomposition of a universal scheme $(\mathcal{U}, C)$, and that it should be logically equivalent to $(\mathcal{U}, C)$. Therefore, the equality of attribute sets $\mathcal{U} = \bigcup_{N_j \in S} R_j$ and the equivalence of integrity constraints $C \equiv \mathcal{I} \cup \left(\bigcup_{N_j \in S} C_j \right)$ should hold.

Let an attribute $A \in \mathcal{U}$ belong to two different relation schemes N_i and N_k from S. It may be supposed that the domains for A in R_i and R_k need not be the same, but both of them should be the subsets of a domain, defined by $\mathcal{Dom}(\mathcal{U}, A)$. Therefore, a domain function Dom_{R_j}: $R_j \rightarrow \mathcal{D}$ is introduced to associate each attribute $A \in R_j$ with a domain from $\mathcal{D}$. Let $\mathcal{Dom}(N_j, A)$ denote a domain D for which $Dom_{R_j}(A, D)$ holds. The function Dom_{R_j} must be specified in such a way that $\mathcal{Dom}(\mathcal{U}, A) \in \mathfrak{Id}(\mathcal{Dom}(N_j, A))$ holds, for each A in any N_j from S. It follows that the logical implication $\mathfrak{id}(\mathcal{Dom}(N_j, A)) \models \mathfrak{id}(\mathcal{Dom}(\mathcal{U}, A))$ must hold. The proof of the statement may be found in [7].

Domain constraint $\mathfrak{id}(\mathcal{Dom}(N_j, A))$ will be called the *domain constraint of an attribute in a relation scheme*. $\mathfrak{id}(\mathcal{Dom}(N_j, A))$ may be validated for an attribute A value for each tuple $t \in Tuple(R_j)$, which is denoted as $\mathfrak{id}(\mathcal{Dom}(\mathcal{U}, A))(t[A])$. A validation is performed according to rules presented in Lemma 1.

2.4. THE TUPLE INTEGRITY CONSTRAINT

Let us first introduce the notions of (i) a null value constraint, and (ii) an attribute value constraint, and then formulate a tuple integrity constraint.

For an attribute A one may expect that null values are allowed in one, but disallowed in an other relation scheme. This possibility can additionally restrict a set of allowed values for an attribute in a relation scheme. Let $\mathcal{Null}(N_j, A)$ be a logical parameter, which denotes a *null value constraint* of an attribute A in a relation scheme N_j. Thus, $\mathcal{Null}(N_i, A) \in \{\bot, \mathsf{T}\}$ holds. If it is $\mathcal{Null}(N_i, A) = \mathsf{T}$, null values for A in an instance over N_j are allowed. Otherwise, if $\mathcal{Null}(N_i, A) = \bot$, null values are disallowed.

In certain cases, a value of $\mathcal{Null}(N_j, A)$ may be a consequence of the appropriate null value constraint $\mathcal{Null}(\mathcal{U}, A)$ in a universal relation scheme only. However, a value of $\mathcal{Null}(N_j, A)$ may also depend of the existence of the other constraints, such as keys.

A null value constraint can be validated for any value from the domain of an attribute $A \in R_j$. Validation of a value d_i, which satisfies $\mathfrak{id}(\mathcal{Dom}(N_j, A))$, is denoted by $\mathcal{Null}(N_j, A)(d_i)$. Validation rules are defined as follows:

- $\mathcal{Null}(N_j, A) = \mathsf{T} \Rightarrow \mathcal{Null}(N_j, A)(d_i) = \mathsf{T}$; and
- $\mathcal{Null}(N_j, A) = \bot \Rightarrow (d_i = \omega \Leftrightarrow \mathcal{Null}(N_j, A)(d_i) = \bot)$.

Specifications of a domain constraint of an attribute in a relation scheme and a null value constraint together build a new integrity constraint type, which is called the attribute value constraint. Let $\tau(N_j, A)$ be an *attribute value constraint*. It is a structure given in the following way:

$$\tau(N_j, A) = (\mathcal{Dom}(N_j, A), \mathcal{Null}(N_j, A)). \quad \textbf{(5)}$$

Both components of $\tau(N_j, A)$ are mandatory.

A validation rule for an attribute value constraint is defined as follows: $\tau(N_j, A)(t[A]) = \mathfrak{id}(\mathcal{Dom}(N_j, A))(t[A]) \wedge \mathcal{Null}(N_j, A)(t[A])$, where $t[A]$ is an A value of a tuple t over the set of attributes R_j. The set of all values satisfying $\tau(N_j, A)$ is denoted as $Domain(N_j, A)$.

Some attribute values from a tuple over R_j may be, in some cases, logically related. This kind of a relationship is modeled by a logical condition, denoted as $\mathcal{Con}(N_j)$. $\mathcal{Con}(N_j)$ is composed of atomic conditions, related by logical operators $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and $\neg$. Each atomic condition is of the form $\alpha_1 \, \theta \, \alpha_2$, where α_1 and α_2 are the terms and $\theta \in \{<, >, \leq, \geq, \neq, =\}$. A term is defined recursively. A term is either (i) a constant $k \in Domain(N_j, A)$, where $A \in R_j$, or (ii) an attribute $A \in R_j$, or (iii) a function $f(\alpha_1,...,\alpha_n)$, where $\alpha_1,...,\alpha_n$ are previously defined terms and f is an n-ary function that is computable on the level of a single tuple over R_j. Each term is obtained by applying aforementioned rules a finite number of times. $\mathcal{Con}(N_j)$ may also be an "empty" condition. In this case, it is marked with a symbol Δ. $\mathcal{Con}(N_j)$, defined in such a way, is called *the logical condition of a relation scheme*. Complete definitions of an atomic condition and a condition $\mathcal{Con}(N_j)$ can be found in [5] and [7].

If a tuple $t \in Tuple(R_j)$ satisfies the attribute value integrity constraint for each $A \in R_j$, then it may be used to validate a logical condition of a relation scheme $Con(N_j)$. A validation is denoted as $Con(N_j)(t)$. Validation is a logical function, i.e. $Con(N_j)(t) \in \{\mathsf{T}, \bot\}$ must hold, for each tuple t from the set $\{t \in Tuple(R_j) \mid (\forall A \in R_j)(\tau(N_j, A)(t[A]) = \mathsf{T})\}$. Validation rules are defined as follows.

- If $Con(N_j) = \Delta$, then $Con(N_j)(t) = \mathsf{T}$.
- If $Con(N_j) \neq \Delta$, then each attribute A in $Con(N_j)$ is replaced by a corresponding value $t[A]$. After that, the values of all the functions in $Con(N_j)$ are calculated. Finally, $Con(N_j)(t) = \mathsf{T}$ will hold iff t satisfies such a transformed condition, where relational and logical operators are validated in a usual way.

Definition 2. A *tuple integrity constraint* $\tau(N_j)$ over a relation scheme N_j is a pair

$$\tau(N_j) = (\{\tau(N_j, A) \mid A \in R_j\}, Con(N_j)), \tag{6}$$

where $\{\tau(N_j, A) \mid A \in R_j\}$ is a set of attribute value constraints and $Con(N_j)$ is a logical condition of a relation scheme. □

A tuple integrity constraint $\tau(N_j)$ may be validated for any tuple t over the set of attributes R_j, which is denoted as $\tau(N_j)(t)$. A validation rule is defined by the logical expression:

$$\tau(N_j)(t) = (\underset{A \in R_j}{\wedge} \tau(N_j, A)(t[A])) \wedge Con(N_j)(t), \tag{7}$$

where $t \in Tuple(R_j)$.

By analyzing the syntax rules used to formulate $Con(N_j)$ and $Con(D)$, it should be noticed that there are examples of logical constraints, which can be equivalently expressed by means of both $Con(N_j)$ and some conditions of the type $Con(Dom(N_j, A))$, $A \in R_j$. In the same way, it is possible to equivalently express the constraint $Null(N_j, A) = \bot$, by incorporating the expression $A \neq \omega$ into $Con(Dom(N_j, A))$ and leaving $Null(N_j, A) = \mathsf{T}$. In order to avoid such ambiguousness, we suppose that $Con(N_j)$ is built in such a way that it expresses only those constraints that could not have been expressed by means of any domain constraint conditions $Con(Dom(N_j, A))$, $A \in R_j$. Besides, we suppose that if $Null(N_j, A) = \mathsf{T}$ holds for an attribute $A \in R_j$, then there must be a tuple $t \in Tuple(R_j)$ for which $Con(N_j)(t) = \mathsf{T}$ and $t[A] = \omega$ hold. If $Con(N_j)$ satisfies both aforementioned conditions, it is called a *well-defined condition*. More details may be found in [7].

Example 2. Let a relation scheme $ORDER(R, C, K_p(R))$ be given, where $R = \{OrdId, Ordate, CustId, Origin, Total\}$, and $C = (\{OrdId\}, \tau(ORDER), \varnothing)$. Let the tuple integrity constraint be defined in the following way:

- $\tau(ORDER) = \tau(N_1) = (\{\tau(N_1, A) \mid A \in R_1\}, CustId > 90000 \Rightarrow Total \geq 10000{,}00)$;
  - $\tau(N_1, OrdId) = (DOrdId, \bot)$, $id(DOrdId) = (Integer, 6, d \geq 1)$;
  - $\tau(N_1, Ordate) = (DOrdate, \bot)$, $id(DOrdate) = (Date, \Delta, \Delta)$;
  - $\tau(N_1, CustId) = (DCOId, \bot)$, $id(DCOId) = (DCustId, \Delta, d \geq 1000)$;
    $id(DCustId) = (Integer, 5, d \geq 1)$;
  - $\tau(N_1, Origin) = (DOrigin, \bot)$, $id(DOrigin) = (Character, 1, d \in \{'d', 'f'\})$;
  - $\tau(N_1, Total) = (DTotal, \bot)$, $id(DTotal) = (Real, \Delta, d \geq 0)$;

The values of attribute *Origin* determine whether a domestic or foreign customer issued a particular order, where *'d'* stands for domestic and *'f'* for foreign. The logical condition $Con(N_1)$ requires that a total amount of an order for a customer with *CustId* grater than *90000* must be grater than or equal *10000,00*. □

2.5. THE UNIQUENESS INTEGRITY CONSTRAINT

Definition 3. Let a relation scheme $N_j(R_j, C_j)$ be given. A *uniqueness integrity constraint* is the expression of the form $\mathcal{Unique}(N_j, X)$, where $X \subseteq R_j$ holds. □

A uniqueness integrity constraint $\mathcal{Unique}(N_j, X)$ may be validated for any relation r_j defined over the attribute set R_j, which is denoted as $\mathcal{Unique}(N_j, X)(r_j)$. Uniqueness integrity constraint is satisfied iff the following two conditions hold:

$$(\forall t_i, t_k \in r_j)((\forall A \in X)(t_i[A] \neq \omega \wedge t_k[A] \neq \omega) \Rightarrow (t_i[X] = t_k[X] \Rightarrow t_i = t_k)), \quad (8)$$

$$(\forall X' \subset X)(\neg(8)). \quad (9)$$

Let the set $\mathcal{K}_j$ of all keys of a relation scheme $N_j(R_j, C_j)$ and a set of attributes $X \subseteq R_j$ be given. According to the definitions of a uniqueness constraint and a key constraint, it follows that the equivalence $(\mathcal{Unique}(N_j, X) \wedge (\forall A \in X)(\mathcal{Null}(N_j, A) = \perp)) \equiv X \in \mathcal{K}_j$ holds ([7]). Hence, we suppose that the set of all relation scheme uniqueness constraints $\mathcal{Uniq}(N_j) = \{X \subseteq R_j \mid \mathcal{Unique}(N_j, X)\}$ contains only those X for which $(\forall K \in \mathcal{K}_j)(K \not\subseteq X)$ holds.

More details, concerning aforementioned constraint types can be found in [5] and [7].

2.6. THE CONSTRAINT SPECIFICATION

The common components of the specification of a constraint $\mathfrak{o} \in \mathcal{I} \cup \left(\bigcup_{N_j \in S} C_j \right)$ are: a constraint type denoted as $\mathcal{ConType}$, where $\mathcal{ConType} \in \{\mathcal{Key}, \mathcal{Unique}, \tau, \mathcal{ReferentialIntegrity}, ...\}$, and a nonempty set of four-tuples:

$$T(\mathfrak{o}) = \{(N_1, \rho_1, At_1, \{(op_1^{i_1}, act_1^{i_1}) \mid i_1 \geq 1\}), ..., (N_m, \rho_m, At_m, \{(op_m^{i_m}, act_m^{i_m}) \mid i_m \geq 1\})\}. \quad (10)$$

In the four-tuple $(N_j, \rho_j, At_j, \{(op_j^{i_j}, act_j^{i_j}) \mid i_j \geq 1\})$, N_j is the name of a relation scheme that is spanned by $\mathfrak{o}$, $\rho_j \in \{referenced, referencing, ...\}$ is the role of N_j in $\mathfrak{o}$, At_j is a set or sequence of attributes from R_j that are *relevant* for $\mathfrak{o}$, and $\{(op_j^{i_j}, act_j^{i_j}) \mid i_j \geq 1\}$ is a set of pairs (*critical operation*, *activity*). An attribute A is relevant for $\mathfrak{o}$ if $\mathfrak{o}$ is used to check values of A. An operation $op_j^{i_j} \in \{insert, delete, update\}$ is a *critical* one if it can violate a constraint and $act_j^{i_j} \in \{NoAction, Cascade, SetDefault, SetNull\}$ is an activity for preserving data consistency. The activity $act_j^{i_j}$ is applied to preserve data consistency during an attempt of the constraint violating by means of $op_j^{i_j}$. Besides, specifications for some constraint types may contain, as the third component, an optional logical condition denoted as $\mathcal{ConstCondition}$.

3. THE SUBSCHEMA

A subschema is designed with respect to a user request and an existing relational database schema. Accordingly, a subschema describes data of a relatively small part of a database, and consists of a set of relation schemes and a set of interrelation constraints. Each relation scheme of a subschema consists of a set of attributes and a set of local constraints. A *role* and a *set of modifiable attributes*, defining possible database update activities, are also assigned to each relation scheme. Each relation scheme of a subschema may be considered as a view on a single base relation scheme. Subschema instances are not materialized.

The goal of a subschema design is to satisfy at least one user request. So, for a database schema there may exist at most as many subschemas as there are identified user requests.

Definition 4. A ***subschema*** is a named pair $P_k(S_k, \mathcal{I}_k)$, where P_k is a subschema name, S_k is a set of relation schemes, and $\mathcal{I}_k$ is a set of interrelation constraints. The set of relation schemes of a subschema P_k is

$$S_k = \{N_i^k(R_i^k, C_i^k, K_p(R_i^k), \mathcal{Role}(P_k, N_i^k), \mathcal{Mod}(P_k, N_i^k), \mathcal{Sr}(P_k, N_i^k)) \mid i \in \{1,..., n\}\}, \quad (11)$$

where N_i^k is a scheme name, R_i^k is an attribute set, C_i^k is a specification of constraints of the form $(\mathcal{K}_i^k, \tau(N_i^k), \mathcal{Uniq}(N_i^k))$, and $K_p(R_i^k)$ is a primary key. $\mathcal{Role}(P_k, N_i^k)$ is a set of relation scheme *roles* in the subschema P_k, and $\mathcal{Mod}(P_k, N_i^k)$ is a set of attributes that may be *modified*. $\mathcal{Sr}(P_k, N_i^k)$ is a function from the set of relation schemes S_k into the set of relation schemes S. □ ([4], [5], [6], [7])

Further explanations of $\mathcal{Sr}(P_k, N_i^k)$, $\mathcal{Role}(P_k, N_i^k)$ and $\mathcal{Mod}(P_k, N_i^k)$ follow.

$\mathcal{Sr}(P_k, N_i^k)$ associates each relation scheme $N_i^k \in S_k$ with a relation scheme $N_j \in S$ for which $R_i^k \subseteq R_j$ holds. The relation scheme N_j will be called the *corresponding* relation scheme for the relation scheme N_i^k. A subschema designer defines the mapping $\mathcal{Sr}(P_k, N_i^k)$. It is supposed that database schema and subschema design guarantee that for each N_i^k in any P_k there is at least one $N_j \in S$ such that $R_i^k \subseteq R_j$ holds.

$\mathcal{Role}(P_k, N_i^k)$ determines the operations that may be performed on an instance of the relation scheme N_i^k. These operations may be built into a transaction program made using the concepts of a subschema P_k. A set of relation scheme roles is a nonempty set, and $\mathcal{Role}(P_k, N_i^k) \subseteq \{r, \imath, m, \eth\}$, where:

- r stands for data reading or referencing;
- $\imath$ stands for data insert;
- m stands for data modification; and
- $\eth$ stands for data deleting.

A subschema P_k is intended for database *querying* only if $(\forall N_i^k \in S_k)(\mathcal{Role}(P_k, N_i^k) = \{r\})$ holds, and it is intended for database *updating* otherwise.

The set $\mathcal{Mod}(P_k, N_i^k)$ contains those attributes of the relation scheme N_i^k that may be modified. If $m \in \mathcal{Role}(P_k, N_i^k)$, then $\mathcal{Mod}(P_k, N_i^k)$ must not equal $\varnothing$.

According to the definition, a subschema is merely a design tool that is needed for a correct transaction program design.

Example 3. Suppose a database schema $(S, \mathcal{I})$ and subschemas $P_1(S_1, \mathcal{I}_1)$, $P_2(S_2, \mathcal{I}_2)$, $P_3(S_3, \mathcal{I}_3)$ are given, where:

- $S = \{ORDER(R_1, C_1, K_p(R_1)), SHIPMENT(R_2, C_2, K_p(R_2)), CUSTOMER(R_3, C_3, K_p(R_3))\}$;
  - $R_1 = \{OrdId, Ordate, CustId, Origin, Total\}$;
  - $C_1 = (\mathcal{K}_1, \tau(N_1), \mathcal{Uniq}(N_1)) = (\{\{OrdId\}\}, \tau(ORDER), \varnothing)$; $K_p(R_1) = \{OrdId\}$;
  - $R_2 = \{ShipId, OrdId, ShipDate, ShipTotal\}$;
  - $C_2 = (\mathcal{K}_2, \tau(N_2), \mathcal{Uniq}(N_2)) = (\{\{ShipId\}\}, \tau(SHIPMENT), \varnothing)$; $K_p(R_2) = \{ShipId\}$;
  - $R_3 = \{CustId, CustName, CustAdrr\}$;
  - $C_3 = (\mathcal{K}_3, \tau(N_3), \mathcal{Uniq}(N_3)) = (\{\{CustId\}\}, \tau(CUSTOMER), \varnothing)$; $K_p(R_3) = \{CustId\})$;
- $\mathcal{I} = \{ORDER[CustId] \subseteq CUSTOMER[CustId], SHIPMENT[OrdId] \subseteq ORDER[OrdId]\}$.
- $S_1 = \{Dom\_Shipped\_Order(R_1^1, C_1^1,...), Shipment(R_1^2, C_1^2,...)\}$;
  - $R_1^1 = \{OrdId, OrDate, CustId, Origin, Total\}$;
  - $C_1^1 = (\mathcal{K}_1^1, \tau(N_1^1), \mathcal{Uniq}(N_1^1)) = (\{\{OrdId\}\}, \tau(Dom\_Shipped\_Order), \varnothing)$; $K_p(R_1^1) = \{OrdId\}$;
  - $R_1^2 = \{ShipId, OrdId, ShipDate, ShipTotal\}$;
  - $C_1^2 = (\mathcal{K}_1^2, \tau(N_1^2), \mathcal{Uniq}(N_1^2)) = (\{\{ShipId\}\}, \tau(Shipment), \varnothing)$; $K_p(R_1^1) = \{ShipId\}$;
  - $\mathcal{Role}(P_1, Dom\_Shipped\_Order) = \mathcal{Role}(P_1, Shipment) = \{r\}$;
  - $\mathcal{Mod}(P_1, Dom\_Shipped\_Order) = \mathcal{Mod}(P_1, Shipment) = \varnothing$;
  - $\mathcal{Sr}(P_1, Dom\_Shipped\_Order) = ORDER$;

- $\mathcal{Sr}(P_1, Shipment) = SHIPMENT$;

- $\mathcal{I}_1 = \{Shipment[OrdId] \subseteq Dom\_Shipped\_Order[OrdId],$
  $Dom\_Shipped\_Order[OrdId] \subseteq Shipment[OrdId]\}$;

- $S_2 = \{Order(R_2^1, \mathcal{C}_2^1,...), Customer(R_2^2, \mathcal{C}_2^2,...)\}$;
  - $R_2^1 = \{OrdId, OrDate, CustId, Origin, Total\}$;
  - $\mathcal{C}_2^1 = (\mathcal{K}_2^1, \tau(N_2^1), \mathcal{U}niq(N_2^1)) = (\{\{OrdId\}\}, \tau(Order), \varnothing)$; $K_p(R_2^1) = \{OrdId\}$;
  - $\mathcal{R}ole(P_2, Order) = \{i, r\}$;
  - $\mathcal{M}od(P_2, Order) = \varnothing$;
  - $\mathcal{Sr}(P_2, Order) = ORDER$;
  - $R_2^2 = \{CustId, CustName, CustAdrr\}$,
  - $\mathcal{C}_2^2 = (\mathcal{K}_2^2, \tau(N_2^2), \mathcal{U}niq(N_2^2)) = (\{\{CustId\}\}, \tau(Customer), \varnothing)$; $K_p(R_2^2) = \{CustId\})$;
  - $\mathcal{R}ole(P_2, Customer) = \{r\}$;
  - $\mathcal{M}od(P_2, Customer) = \varnothing$;
  - $\mathcal{Sr}(P_2, Customer) = CUSTOMER$;
- $\mathcal{I}_2 = \{Order[CustId] \subseteq Customer[CustId]\}$;

- $S_3 = \{Order(R_3^1, \mathcal{C}_3^1,...), Shipment(R_3^2, \mathcal{C}_3^2,...)\}$;
  - $R_3^1 = \{OrdId, OrDate, CustId, Origin, Total\}$;
  - $\mathcal{C}_3^1 = (\mathcal{K}_3^1, \tau(N_3^1), \mathcal{U}niq(N_3^1)) = (\{\{OrdId\}\}, \tau(Order), \varnothing)$; $K_p(R_3^1) = \{OrdId\}$;
  - $\mathcal{R}ole(P_3, Order) = \{r\}$;
  - $\mathcal{M}od(P_3, Order) = \varnothing$;
  - $\mathcal{Sr}(P_3, Order) = ORDER$;
  - $R_3^2 = \{ShipId, OrdId, ShipDate, ShipTotal\}$;
  - $\mathcal{C}_3^2 = (\mathcal{K}_3^2, \tau(N_3^2), \mathcal{U}niq(N_3^2)) = (\{\{ShipId\}\}, \tau(Shipment), \varnothing)$; $K_p(R_3^2) = \{ShipId\}$;
  - $\mathcal{R}ole(P_3, Shipment) = \{i, r\}$;
  - $\mathcal{M}od(P_3, Shipment) = \varnothing$;
  - $\mathcal{Sr}(P_3, Shipment) = SHIPMENT$;
- $\mathcal{I}_3 = \{Shipment[OrdId] \subseteq Order[OrdId]\}$;

Subschema P_1 is aimed for data queries only, P_2 is aimed for order entry, and P_3 is aimed for entry of shipments that are initiated by orders. Subschema P_1 is associated with a business unit whose task is to control domestic orders. In this business unit users are not interested in customer data except for customer id number. The domain constraint of attribute *Origin* in subschema P_1 should be more restrictive than the corresponding constraint in the database schema, because only the orders of domestic customers are required. P_1 also contains the inclusion dependency $Dom\_Shipped\_Order[OrdId] \subseteq Shipment[OrdId]$ that forces selecting only those order tuples from a relation over *ORDER* that are referenced by some shipment tuples.

Suppose all the tuple integrity constraints that are embedded into relation schemes whose name has the same meaning are the same, except the domain constraint of attribute *Origin.* Let a tuple integrity constraint $\tau(ORDER)$ be defined as in Example 2, and a tuple integrity constraint of scheme *Dom_Shipped_Order* be defined as follows:

- $\tau(Dom\_Shipped\_Order) = \tau(N_1^1) = (\{\tau(N_1^1, A) \mid A \in R_1^1\}, CustId > 90000 \Rightarrow Total \geq 10000{,}00)$;
  - $\tau(N_1^1, OrdId) = \tau(N_1, OrdId)$;
  - $\tau(N_1^1, Ordate) = \tau(N_1, Ordate)$;
  - $\tau(N_1^1, CustId) = \tau(N_1, CustId)$;
  - $\tau(N_1^1, Origin) = (DDOrigin, \perp)$, $\dot{\imath}\mathfrak{d}(DDOrigin) = (DOrigin, \Delta, d='d'\})$;
  - $\tau(N_1^1, Total) = \tau(N_1, Total)$;

In subschema P_1, $Domain(Dom\_Shipped\_Order, Origin) = \{'d'\}$ holds, whereas in the relation scheme *Order* of P_2, P_3 so as in the relation scheme *ORDER* of the database schema, $Domain(Order, Origin) = Domain(ORDER, Origin) = \{'d', 'f'\}$ holds. □

4. XML SPECIFICATION OF A SUBSCHEMA

The notion of a subschema and its formal structure are introduced in Section 3. That structure can be formally expressed by means of XML syntax [9]. An XML DTD formal specification of a subschema structure is shown in Figure 2. It is used to syntactically validate any automatically generated XML specification of subschema. The subschema concepts and the corresponding DTD concepts are shown in Table 1 in order to help in understanding the specification shown in Figure 2.

The Subschema Concept	Name of the DTD Element	Name of the DTD attribute
$P_k(S_k, \mathcal{I}_k)$	subschema	subschName
$o \in \mathcal{I}_k$, $T(o)$	subschConstraint	conName, conClass
$N_i^k(R_i^k, C_i^k, ..., Sr(P_k, N_i^k)) \in S_k$	relScheme	schemeName, dbSchemeName
$o \in C_i^k$, $T(o)$, $K_p(R_i^k)$	schemeConstraint	conName, conClass
$x \in Role(P_k, N_i^k)$	role	roleID
$A \in R_i^k$, $Dom(N_i^k, A)$, $Mod(P_k, N_i^k)$	attribute	attName, attMod
$D_p \in \mathcal{D}p$, $id(D_p)$	predefDomain	type, maxLength
$D \in \mathcal{D} \setminus \mathcal{D}p$, $id(D)$	domain	length
$Con(D)$	domCondition	-
$o \in C_i^k$, $T(o)$, $id(Dom(N_i^k, A))$, $K_p(R_i^k)$	attrConstraint	conName, conClass
$(N_j, \rho_j, At_j, \{(op_j^{ij}, act_j^{ij}) \mid i_j \geq 1\})$	consRelScheme	schemeName, schemeRole
$A \in At_j$	conSchAttr	conAttName
(op_j^{ij}, act_j^{ij})	conValActivity	-
op_j^{ij}	operation	operType
act_j^{ij}	action	actType
$Con(D)$	attCond	-
$Con(N_i^k)$	schCond	-
ConstCondition for an $o \in \mathcal{I}_k$	sbcCond	-
-	validateTime	validateTimeType

Table 1. The List of Subschema Concepts with Its XML DTD Equivalents

The XML DTD specification of a subschema is designed in a way to reach a compromise between two goals. The first one was to support all the concepts, introduced in Sections 2 and 3. The other one was to support such a subschema specification that would be in accordance with the syntax of ANSI SQL92 standard [2]. This way, DTD shown in Figure 2 is somehow more general than the specifications introduced in Sections 2 and 3. For example, it enables, in accordance with SQL92, defining of validation time for each subschema constraint (see the element *ValidateTime* in DTD). Also, it generally offers more than one way to specify the same subschema constraint using SQL92 syntax. For example, if a key consists of only one attribute, the corresponding key constraint can be specified either at the relation scheme level, using the element *schemeConstraint*, or at the attribute level, using the element *attrConstraint*.

However, DTD shown in Figure 2 is, in some details, less restrictive than the structuring rules for subschema given in Section 3, and the specifications of various types of constraints, given in Section 2. To illustrate the claim, let us consider a *CHECK* constraint that implements the logical condition of a tuple constraint. It may be defined within the element *schemeConstraint* specifying a value 'CHECK' of the attribute *conClass*. However, the condition *schCond* may be omitted, despite the fact that it should be defined. Oppositely, if a primary key is defined using the element *schemeConstraint*, then *schCond* must not be defined, but the DTD syntax does not cover this rule. Next, note that the syntax rules for $Con(D)$ and $Con(N_j)$ are not embedded into the DTD, since the elements *domCondition*, *attCond* and *schCond* have been left unstructured. Embedding the syntax rules for $Con(D)$ and $Con(N_j)$ into the DTD represent a further research task. We believe that it would be very difficult and practically even unnecessary to cover all the formal rules by means of a DTD. Besides, it is expected that almost all structuring rules should be embedded into a dictionary of the selected CASE tool.

```
<!DOCTYPE subschema [
      <!ELEMENT subschema (relScheme+, subschConstraint*)>
      <!ELEMENT relScheme (role+, attribute+, schemeConstraint*)>
      <!ATTLIST relScheme
            schemeName CDATA #REQUIRED>
            dbSchemeName CDATA #REQUIRED>
      <!ELEMENT role EMPTY>
      <!ATTLIST role
            roleID (read|insert|modify|delete) #REQUIRED>
      <!ELEMENT attribute ((domain|predefDomain), attrConstraint*)>
      <!ATTLIST attribute
            attName CDATA #REQUIRED
            attMod (yes|no) #REQUIRED>
      <!ELEMENT domain ((predefDomain|domain), domCondition)>
      <!ELEMENT predefDomain EMPTY >
      <!ATTLIST predefDomain
            type (STRING|INTEGER|REAL|DECIMAL|BOOLEAN|DATE|CDATA) #REQUIRED
            maxLength CDATA "">
      <!ATTLIST domain
            length CDATA "">
      <!ELEMENT domCondition (#PCDATA)>
      <!ELEMENT attrConstraint (consRelScheme+,attCond?, validateTime)>
      <!ELEMENT schemeConstraint (consRelScheme+,schCond?,validateTime)>
      <!ELEMENT subschConstraint (consRelScheme+,sbcCond?,validateTime)>
      <!ELEMENT consRelScheme (conSchAttr+, conValActivity+)>
      <!ATTLIST consRelScheme
            schemeName CDATA #REQUIRED
            schemeRole (referencing|referenced) #REQUIRED>
      <!ELEMENT conSchAttr EMPTY>
      <!ATTLIST conSchAtt
            conAttName CDATA #REQUIRED>
      <!ELEMENT conValActivity (operation, action)>
      <!ELEMENT operation EMPTY>
      <!ATTLIST operation
            operType (ON_UPDATE|ON_DELETE|ON_INSERT) #REQUIRED>
      <!ELEMENT action EMPTY>
      <!ATTLIST action
            actType (NO_ACTION|CASCADE|SET_DEFAULT|SET_NULL) #REQUIRED>
      <!ELEMENT attCond (#PCDATA)>
      <!ELEMENT schCond (#PCDATA)>
      <!ELEMENT sbcCond (#PCDATA)>
      <!ELEMENT validateTime EMPTY>
      <!ATTLIST validateTime
            validateTimeType  (     INITIALY_DEFERRED|
                                    INITIALY_IMMEDIATE_NOT_DEFERRABLE|
                                    INITIALY_IMMEDIATE_DEFERRABLE)
                              ) "INITIALY_IMMEDIATE_NOT_DEFERRABLE">
      <!ATTLIST attrConstraint
            conName CDATA #REQUIRED
            conClass(NOTNULL|CHECK|PRIMARY_KEY|UNIQUE|REFERENCES)#REQUIRED>
      <!ATTLIST schemeConstraint
            conName CDATA #REQUIRED
            conClass (CHECK|PRIMARY_KEY|UNIQUE|REFERENCES) #REQUIRED>
      <!ATTLIST subschConstraint
            conName CDATA #REQUIRED
            conClass (ASSERTION) #REQUIRED>
      <!ATTLIST subschema
            subschName CDATA #REQUIRED>
]>
```

Figure 2. XML DTD Specification of The Subschema Structure

For the purpose of the paper, we have chosen a DTD concept instead of more flexible XML Schema concept to express the subschema structuring rules. The only reason for it is that an equivalent XML Schema specification is too complex to be presented in the paper in detail.

5. CONCLUSION

In this paper, the notion of a subschema that represents the data definition part of a transaction program specification is introduced. The common constraint types that are used to specify a subschema constraints are also presented in a formal way. A subschema is a structure, expressed by means of relational database schema concepts, extended by the specification of the allowed database update activities, and a mapping that uniquely bonds subschema relation schemes with the database relation schemes. A formal way to describe the structure of a subschema by means of XML is presented in the paper. This description supports SQL subschema specification using concepts of ANSI SQL92 standard.

We have chosen the concepts of the relational data model to define the notion of a subschema, since we needed its powerful mathematical formalism to express our ideas and solutions in a precise way. But we believe that our results may be applied to other data models, too. We propose that subschema, as a part of a transaction program should have all the necessary constraints defined, despite the fact that all database constraints should be implemented and validated under a DBMS. By embedding all the relevant database constraints into a subschema, we obtain a possibility to validate them earlier than a DBMS, i.e. just after a user action which may cause a constraint violation. It could be one of the ways to make a user interface more "friendly". In order to reach this goal, it is necessary to provide formal rules, which will guarantee that the set of subschema constraints will imply all those database schema constraints that might be violated by the allowed database update operations.

The idea of using code generators to produce transaction programs automatically is not a new one. One of the problems concerning this idea is that many of the current code generators are highly specialized for only one programming environment. Using such generators makes application software highly dependent on the chosen programming environment. Our main idea is to build a code generator, as a part of an integrated CASE tool, which can produce executable program specifications that are mostly independent of a particular programming environment. We propose using of XML, as a specification language for this purpose [3]. In this paper, we have represented the formally defined structure of a subschema by means of XML. One of the open questions is how to formalize and precisely specify by means of XML the other constraint types, not mentioned in the paper, but used in the software design practice.

6. REFERENCES

[1] Codd, E. F. (1990): *The Relational Model for Database Management Version 2*, Addison-Wesley-Publishing-Company, USA

[2] Date, C. J. (1994): *A Guide to the SQL Standard*, Addison-Wesley Publishing Company, USA

[3] Govedarica, M. (2001): *An Automated Development of Information System Application Prototypes*, PhD Thesis, University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Yugoslavia

[4] Luković, I.; Mogin P. (2000): *On The Role of Subschema as A Component of The Implementation Specification of A Program*, VI Symposium on Computer Science and Information Technologies YUINFO, Kopaonik, Yugoslavia, Proceedings on CD ROM

[5] Mogin, P.; Luković, I.; Govedarica M. (2000): *Database Design Principles*, University of Novi Sad, Faculty of Technical Sciences & MP "Stylos", Novi Sad, Yugoslavia

[6] Mogin, P.; Luković, I. (1999): *An Approach to Database Design*, International Journal of INDUSTRIAL SYSTEMS, Vol. 1, No. 2, Novi Sad, Yugoslavia 59-68

[7] Ristić, S. (in progress): *A Research of Subschema Consolidation Problem*, PhD Thesis, University of Novi Sad, Faculty of Economics, Subotica, Yugoslavia

[8] *UIML / User Interface Markup Language – Draft Specification,Version 2.0* Harmonia Inc, 2000

[9] *XML / Extensible Markup Language*, http://www.w3.org/XML/

[10] *XSL / Extensible Stylesheet Language*, http://www.w3.org/TR/xsl/