BILINGUAL PUBLISHING CO.
Pioneer of Global Academics Since 1984

Journal of Electronic & Information Systems
https://ojs.bilpublishing.com/index.php/jeis

ARTICLE

On Detecting and Enforcing the Non-Relational Constraints Associated to Dyadic Relations in *MatBase*

Christian Mancas*

Mathematics and Computer Science Dept., Ovidius University, Constanta, Romania

ARTICLE INFO

Article history
Received: 19 June 2020
Accepted: 28 August 2020
Published Online: 31 October 2020

Keywords:
Conceptual data modeling
Database constraints theory
Non-relational constraints
Data structures and algorithms for data management
Dyadic relation properties
Data quality
(Elementary) Mathematical Data Model
MatBase

ABSTRACT

MatBase is a prototype data and knowledge base management expert intelligent system based on the Relational, Entity-Relationship, and (Elementary) Mathematical Data Models. Dyadic relationships are quite common in data modeling. Besides their relational-type constraints, they often exhibit mathematical properties that are not covered by the Relational Data Model. This paper presents and discusses the *MatBase* algorithm that assists database designers in discovering all non-relational constraints associated to them, as well as its algorithm for enforcing them, thus providing a significantly higher degree of data quality.

1. Introduction

MatBase [10-13] is a prototype data and knowledge base management expert intelligent system based on the Relational (RDM) [1,3,6], the Entity-Relationship (E-RDM) [2,6,18], and the (Elementary) Mathematical ((E)MDM) [10,13] Data Models, as well as on Datalog [1,13], already successfully used for years by a couple of software developing companies, as well as in our University Database lectures and labs. Currently, there are two implementations of *MatBase*: one mainly for University labs developed in MS Access and one mainly for the IT industry developed in C# and MS SQL Server.

Any (conventional) database (db) scheme is a triple <*S, M, C*>, where *S* is a non-void finite collection of sets, *M* a finite non-void set of mappings defined on and taking values from sets in *S*, and *C* a similar one of constraints (i.e. closed first-order predicate calculus with equality formulas [17]) over the sets in *S* and mappings in *M*. Sets and mappings constitute the structure of dbs, while constraints, which are formalizing business rules, are meant to allow storing only plausible data into them.

In RDM, the sets of *S* are tables and views, the mappings of *M* are their columns, and the constraints of *C*

**Corresponding Author:*
Christian Mancas,
Mathematics and Computer Science Dept., Ovidius University, Constanta, Romania;
Email: christian.mancas@gmail.com

are incorporated in the table schemes. Unfortunately, both RDM and E-RDM have only a handful of constraint types, which are not at all enough to guarantee data plausibility. Most of the RDM Management Systems (RDBMS) provide only 6 types of (relational) constraints, namely: domain (range), not-null, default value, key (uniqueness), foreign key (referential integrity), and check (tuple).

For example, using only these 6 types of constraints, even a very simple table (storing automatically generated id numbers in column *P_Id*, First and Last Names, Birth and Passed Away Dates, Sex, and parents of some people of interest, having foreign keys *Mother* and *Father*, both referencing *P_Id*, and not allowing persons to die before or more than 160 years after being born) like the one in Figure 1 might still store highly implausible data.

PERSONS (BDate* + 160years ≥ *PADate* ≥ *BDate)

P_Id	*FName*	*LName*	*BDate*	*Sex*	*Mother*	*Father*	*PADate*
auton	ASCII (64)	ASCII (64)	[1/1/1900, *today*]	{'F', 'M'}	in *P_Id*	in *P_Id*	[1/1/1900, *today*]
not-null	not-null	not-null	not-null	not-null			
1	John	Smith	1/1/2020	M	4	1	1/1/2020
2	Mary	Jane	1/1/1900	F	4	5	1/12/1999
3	Paul	Smith	1/1/2010	M	1	2	1/1/2012
4	Anne	Jane	1/1/2020	F	4	3	
5	Peter	Smith	1/1/1990	M	2	2	

Figure 1. A table with relationally valid, but highly implausible data

It is quite simple to check that data in this table does not violate any of its relational constraints (5 of domain type, 5 not-nulls, 1 key, 2 foreign keys, 1 check/tuple). However, this data is highly implausible, because, according to it:

(1) John Smith is his own father and has a mother born same day as him.

(2) Mary Jane has a mother born after she died, a father born 90 years after her birth, and is the father of her father.

(3) Paul Smith has a man (born after him) as mother and a woman (who died 20 years before his birth) as father.

(4) Anne Jane is her own mother and has a father who died 8 years before her birth.

(5) Peter Smith has a woman as both father and mother, was born 90 years after her birth, and is the father of his father.

In (E)MDM, in order to prevent storing such implausible data, the following 6 non-relational constraints should be added to the above 14 relational ones, in the scheme of this table:

(1) C_1: ($\forall x \in$ *PERSONS*)(*Sex*(*Mother*(x)) = 'F') (i.e. only women may be mothers)

(2) C_2: ($\forall x \in$ *PERSONS*)(*Sex*(*Father*(x)) = 'M') (i.e. only men may be fathers)

(3) C_3: *Mother* acyclic, C_4: *Father* acyclic (i.e. nobody may be her/his own mother/father, neither directly, nor indirectly)

(4) C_5: ($\forall x \in$ *PERSONS*)(*BDate*(x) ≤ *PADate*(*Mother*(x)) ∧ *BDate*(x) + 5years ≤ *BDate*(*Mother*(x)) ≤ *BDate*(x) + 75years) (i.e. no mother may give birth after her death, before being 5 years old, or after being 75 years old)

(5) C_6: ($\forall x \in$ *PERSONS*)(*BDate*(x) ≤ *PADate*(*Father*(x)) + 10month ∧ *BDate*(x) + 9years ≤ *BDate*(*Father*(x)) ≤ *BDate*(x) + 100years) (i.e. no father may have a child after 10 months from his death, before being 9 years old, or after being 100 years old)

As proved by this example (as well as many others, see, e.g. [6,7,13]), when a single business rule is not formalized by a corresponding constraint and/or that constraint is not enforced in the corresponding db scheme, implausible data may be stored in that db.

Just like for object sets and mappings between them, constraints can be discovered only by humans. However, computer science and math can assist in this process: e.g. the keys discovery assistance algorithms [6,8,13], the algorithm for assisting discovery of non-relational constraints associated to the E-RDM diagram cycles [11,13], the similar ones for endofunctions and object constraints [12,13], the constraint sets coherence and minimality ones [10,13], etc.

Currently, non-relational constraints are, unfortunately, not discovered by db architects, who are not even aware of their typology. Some of them are considered by some software architects, sometimes, but always in an ad-hoc manner. Most of them are ignored, so not enforced, reported then as bugs by customers, and enforced as software fixes, most of the times too late, after corresponding db instances are seriously polluted with implausible data.

This paper introduces and discusses first an algorithm for assisting discovery of non-relational constraints associated to dyadic relations, providing a valuable tool to db and software architects. By replacing in it dyadic relationships with tables and mappings with columns, one may successfully use it for corresponding RDM table schemes as well (corresponding table schemes should have a concatenated key made out of two not null foreign keys referencing a same other table).

This algorithm is embedded in *MatBase*, which is also automatically generating code for enforcing all non-relational constraint types provided by (E)MDM, thus significantly enhancing productivity and quality of db applications development. This paper also presents and discusses the algorithm for enforcing the constraints associated to dyadic relationships.

This first section presents an overview of the 73 (E) MDM constraint types, related work, and the paper outline.

1.1 (E)MDM Constraint Types

(E)MDM provides three constraint categories: set, mapping, and object [10,13].

The set category has two subcategories (and 16 types):

(1) general set (comprising five types: inclusion, set equality, disjointness, union, and direct sum) and

(2) dyadic relation (comprising eleven types: connectivity, reflexivity, irreflexivity, symmetry, asymmetry, transitivity, intransitivity, Euclideanity, inEuclideanity, acyclicity, equivalence).

The mapping category has five categories (and 56 types):

(1) General function (having six types: totality, nonprimeness, one-to-oneness, ontoness, bijectivity, and default value),

(2) Endofunction (having thirteen types: reflexivity, null-reflexivity, irreflexivity, symmetry, null-symmetry, asymmetry, idempotency, null-idempotency, anti-idempotency, equivalence, null-equivalence, acyclicity, canonical surjection),

(3) Function product (having three types: existence, nonexistence, and minimal one-to-oneness),

(4) Homogeneous binary function product (having eighteen types: connectivity, null-connectivity, reflexivity, null-reflexivity, null-identity, irreflexivity, symmetry, null-symmetry, asymmetry, transitivity, null-transitivity, intransitivity, Euclideanity, null-Euclideanity, inEuclideanity, acyclicity, equivalence, null-equivalence), and

(5) Function diagram (having sixteen types: commutativity, null-commutativity, anti-commutativity, generalized commutativity, local commutativity, local null-commutativity, local anti-commutativity, local symmetry, local null-symmetry, local asymmetry, local idempotency, local null-idempotency, local anti-idempotency, local equivalence, local null-equivalence, local acyclicity).

The object constraints are generalizing RDM tuple (check) constraints, being closed Horn clauses [17] (e.g. C_1, C_2, C_5, and C_6 above).

A *generalized commutativity* constraint is an object constrained associated to a function diagram (i.e. implying only sets and mappings from that diagram).

Please recall that dyadic relations are binary homogeneous ones (e.g. $PREREQUISITES \subseteq COURSES^2$), endofunctions (autofunctions, self-functions) have same domain and codomain (e.g. $Mother : PERSONS \rightarrow PERSONS$), and homogeneous binary function products (hbfp) are of the type $f \bullet g : D \rightarrow C^2$ (e.g. $EmbarkmentAirport \bullet DestinationAirport : BOARDING\_PASSES \rightarrow AIRPORTS^2$).

For the differences between mathematical relations and db relationships see [9]. In (E)MDM, dyadic relationships are denoted $R = (f \rightarrow T, g \rightarrow T)$, where f and g are R's roles (i.e. mathematically, the corresponding canonical Cartesian product projections).

For example, the dyadic relationship $PREREQUISITES = (Prerequisite \rightarrow COURSES, Course \rightarrow COURSES)$ should always be acyclic: otherwise, no student might ever enroll in any of the courses involved in a cycle.

Any dyadic relationship has the following 5 relational constraints: both f and g are not null and foreign keys referencing table T, and $f \bullet g$ is a key.

Object constraints may be as well associated to dyadic relations (e.g. $(\forall x \in PREREQUISITES)(\forall y,z \in ENROLLMENTS)(Student(y) = Student(z) \wedge Course(y) = Prerequisite(x) \wedge Course(z) = Course(x) \Rightarrow EnrollDate(z) > CompletionDate(y))$, i.e. no student may enroll to a course before completion of all of its prerequisites) and to endofunctions [12,13].

A function is *total* if it does not take null values (i.e. it is totally defined or, equivalently, its codomain is disjoint from the NULLS distinguished set of null values) and is *nonprime* if it cannot be part of any key (i.e. not only not one-to-one, but not a member of any minimally one-to-one function product; e.g. *Height, Length, Width, Color,* etc.).

An endofunction or a hbfp that may take null values and has property P (e.g. reflexivity, symmetry, etc.) for all of its not null values is said to have property *null-P* (e.g. $ReplacementPart : PART\_TYPES \rightarrow PART\_TYPES$ is null-idempotent, because any replacement part type is replaced by itself, but there are part types that may not be replaced by other ones).

(E)MDM extended the RDM *existence* constraints $f \vdash g$ ("whenever f takes not null values, g must take not null values as well") by allowing both f and/or g to be computed functions (e.g. $e\text{-}mail \vdash City \bullet Address$, i.e. whenever the e-mail address of someone is known to the db, then both the city and the address within the city where he/she lives should also be known to the db).

A *nonexistence* constraint, denoted $\neg \vdash f_1 \bullet \ldots \bullet f_n$, should be read "at most one of the $f_1, \ldots, f_n$ may be not null for any x" (e.g. $\neg\vdash$ *TributaryTo* • *Lake* • *Sea* • *Ocean* • *LostInto* : *RIVERS* → *RIVERS* × *LAKES* × *SEAS* × *OCEANS* × *GEOGRAPHIC_UNITS* formalizes the constraint "a river may empty in only one place, be it another river, a lake, a sea, an ocean, or another geographic unit type, e.g. a desert").

A function diagram made of mappings *f, g* : *D* → *C* (which may be atomic or composite functions) *anti-commutes* if $(\forall x \in D)(f(x) \neq g(x))$ (e.g. $(\forall x \in BOARDING\_PASSES)(EmbarkmentAirport(x) \neq DestinationAirport(x))$).

A function diagram has a *local property P* (e.g. symmetry, idempotency, etc.) in one of its sets if it is of the circular type [11,13] and the composite endofunction defined on and taking values from that set has the property *P* (e.g. the diagram made of the functions *Department* : *EMPLOYEES* → *DEPARTMENTS* and *Manager* : *DEPARTMENTS* → *EMPLOYEES* is locally idempotent in *EMPLOYEES*, because the composite endofunction *Manager* ° *Department* : *EMPLOYEES* → *EMPLOYEES* is idempotent, i.e. any manager works in the department that he/she manages).

1.2 Related Work

Theoretically, the (E)MDM is based on the semi-naïve theory of sets, relations, and functions [5], as well as on the first order logic [14].

Other mathematical data models are the categorial [17] and graph [4,16] ones.

Generally, there are very few db research results on non-relational constraints (e.g. [18]), except for the (E) MDM related ones ([10-13]).

MatBase algorithm for enforcing object constraints is presented in [12].

Dyadic relations were extensively studied within the realm of the set theory (e.g. [5]).

The most closely related approaches are based on business rules management (BRM) [15] and their corresponding implemented systems (BRMS) and process managers (BPM). From this perspective, *MatBase* is also a BRMS, but a formal, automatically code generating one.

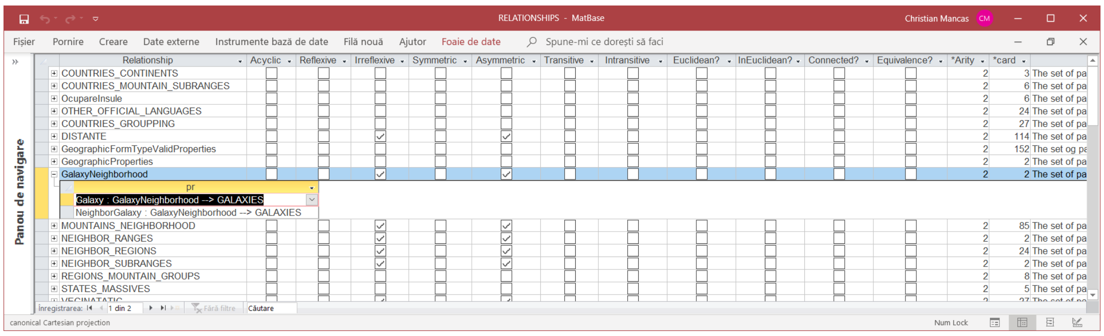
A somewhat related approach as well is the logical constraint programming [19], aimed only at solving polynomial-complexity combinatoric problems (e.g. planning, scheduling, etc.).

1.3 Paper Outline

Section 2 concisely presents the *MatBase* graphical user interface (GUI) for managing relationships (which includes the subset of dyadic ones). Sections 3 is devoted to the *MatBase* algorithm for assisting discovery of dyadic relationship constraints. Section 4 presents the *MatBase* algorithm for enforcing the non-relational constraints associated to dyadic relationships. Section 5 discusses the complexity, optimality, and usefulness of these two algorithms. The paper ends with conclusions and references.

2. *MatBase* GUI for Relationships

Figure 2 shows the *RELATIONSHIPS* form that can be open from the *MatBase MetaCatalog / Scheme Updates / EMDM Scheme / Sets / Views* submenu. This form (together with its embedded subform) is the GUI for managing the db relationships' schemas.



Relationship	Acyclic	Reflexive	Irreflexive	Symmetric	Asymmetric	Transitive	Intransitive	Euclidean?	InEuclidean?	Connected?	Equivalence?	*Arity	*card	
COUNTRIES_CONTINENTS	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	3	The set of pa
COUNTRIES_MOUNTAIN_SUBRANGES	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	6	The set of pa
OcupareInsule	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	6	The set of pa
OTHER_OFFICIAL_LANGUAGES	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	24	The set of pa
COUNTRIES_GROUPPING	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	27	The set of pa
DISTANTE	☐	☐	✓	☐	✓	☐	☐	☐	☐	☐	☐	2	114	The set of pa
GeographicFormTypeValidProperties	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	152	The set og pa
GeographicProperties	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	2	The set of pa
GalaxyNeighborhood	☐	☐	✓	☐	✓	☐	☐	☐	☐	☐	☐	2	2	The set of pa
MOUNTAINS_NEIGHBORHOOD	☐	☐	✓	☐	✓	☐	☐	☐	☐	☐	☐	2	85	The set of pa
NEIGHBOR_RANGES	☐	☐	✓	☐	✓	☐	☐	☐	☐	☐	☐	2	2	The set of pa
NEIGHBOR_REGIONS	☐	☐	✓	☐	✓	☐	☐	☐	☐	☐	☐	2	24	The set of pa
NEIGHBOR_SUBRANGES	☐	☐	✓	☐	✓	☐	☐	☐	☐	☐	☐	2	2	The set of pa
REGIONS_MOUNTAIN_GROUPS	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	8	The set of pa
STATES_MASSIVES	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	2	5	The set of pa

pr
Galaxy : GalaxyNeighborhood --> GALAXIES
NeighborGalaxy : GalaxyNeighborhood --> GALAXIES

Figure 2. MatBase RELATIONSHIPS form and its subform

Besides the columns visible in this figure (with the computed **Arity* -displaying the number of roles- and **card* -showing the cardinality of the instances-), there are other ones as well: an optional *Description* (whose left end is visible in Figure 2), the required *Database* (to which relationships belong), a *System* flag (to distinguish between user and *MatBase* metacatalog relationships), etc.

Users may not only inspect the set of all relationships managed by the system, but also delete them (if they are not underlying sets for hierarchically higher order relationships), insert new relationships and/or modify existing ones.

For example, the currently selected row in Figure 2 corresponds to the dyadic relationship *GalaxyNeighborhood* = (*Galaxy* → *GALAXIES*, *NeighborGalaxy* → *GALAXIES*) from the *Geography* db, which has been declared to be irreflexive (as no galaxy is its own neighbor) and asymmetric (as, whenever galaxy x is neighbor to galaxy y, it does not make sense to also store the fact that *y* is neighbor to *x*).

The embedded subform manages the roles (i.e. the canonical Cartesian projections) of the relationships.

The checkboxes from *Acyclic*? to *Equivalence*? are active only for the dyadic relationships. Whenever you uncheck one of them and then you confirm your request, *MatBase* either rejects it if it is a redundant one (e.g. if that relationship is acyclic, you cannot uncheck either asymmetry or irreflexivity, as both are implied by acyclicity) or is removing the automatically generated code for enforcing the corresponding constraint.

Whenever you check one of them and then you confirm your request, *MatBase* first checks whether the newly resulted constraint set is coherent [10,13] and rejects it if this is not the case (e.g. you cannot declare a relationship being both acyclic and symmetric); then, it checks whether the data instance of that relationship satisfies the desired constraint and rejects it if this is not the case (e.g. you cannot declare a relationship as being irreflexive as long as there is a pair of the type <*x, x*> in its data instance); finally, if everything is ok, *MatBase* accepts the new constraint and automatically generates the code for enforcing it; moreover, it also automatically remove the code for enforcing constraints that became redundant (e.g. if the relationship was irreflexive and becomes acyclic, irreflexivity enforcing code is removed) and checks all corresponding implied constraints.

Whenever you double-click on the row corresponding to a dyadic relationship and confirm your request, *MatBase* launches the algorithm presented in the next section, which assists users in the process of discovering all non-relational constraints associated with that relationship.

3. *MatBase* Algorithm for Assisting Discovery of Dyadic Relationship Constraints

Figures 3 and 4 present the Algorithm *ADDRC*, designed for assisting discovery of dyadic relationship constraints, which is implemented in both *MatBase* versions.

This Algorithm is optimally designed by incorporating the following mathematical results on dyadic relation properties [5,10,13]:

✓ acyclicity $\Rightarrow$ asymmetry

✓ asymmetry $\vee$ intransitivity $\vee$ inEuclideanity $\Rightarrow$ irreflexivity

✓ asymmetry $\wedge$ transitivity $\Rightarrow$ acyclicity

✓ reflexivity $\wedge$ Euclideanity $\Rightarrow$ symmetry $\wedge$ transitivity

✓ symmetry $\wedge$ Euclideanity $\Rightarrow$ transitivity

✓ symmetry $\wedge$ inEuclideanity $\Rightarrow$ intransitivity

✓ symmetry $\wedge$ transitivity $\Rightarrow$ Euclideanity

✓ symmetry $\wedge$ intransitivity $\Rightarrow$ inEuclideanity

✓ irreflexivity $\wedge$ transitivity $\Rightarrow$ asymmetry

✓ symmetry $\wedge$ intransitivity $\wedge$ inEuclideanity $\Rightarrow$ $\neg$connectivity

ALGORITHM *ADDRC.* Dyadic Relationship Constraints Discovery Assistance

```
Input: a db scheme S and one of its dyadic relationship R
Output: S augmented with the newly discovered constraints associated to R (if any)
Strategy:
  if R is, could and should be asymmetric then addCnstr(R, "asymmetric") else
  if R is, could and should be symmetric then addCnstr(R, "symmetric") end if;
  if R is, could and should be irreflexive then addCnstr(R, "irreflexive") else
  if R is, could and should be reflexive then addCnstr(R, "reflexive") end if;
  if R is, could and should be acyclic then addCnstr(R, "acyclic") end if;
  if R is, could and should be transitive then addCnstr(R, "transitive") else
  if R is, could and should be intransitive then addCnstr(R, "intransitive") end if;
  if R is, could and should be Euclidean then addCnstr(R, "Euclidean") else
  if R is, could and should be inEuclidean then addCnstr(R, "inEuclidean") end if;
  if R is, could and should be connected then addCnstr(R, "connected") end if;
End ALGORITHM ADDRC;
```

Figure 3. Algorithm ADDRC (Dyadic Relationship Constraints Discovery Assistance)

ALGORITHM *addCnstr(R, C).* Adds constraint *C* to dyadic relationship *R*'s scheme

```
Input: a dyadic relationship R of db scheme S and a constraint type C
Output: S augmented with C for R and corresponding redundant constraints (if any)
Strategy:
  $S = S \cup \{R\ C\}$;
  Add corresponding redundant constraints for R in S;
End ALGORITHM addCnstr;
```

Figure 4. Algorithm addCnstr

For any dyadic relationship *R*, "is *C*" means that its data instance satisfies constraint (i.e. mathematical property) *C*; "could be *C*" means that by adding constraint *C* to its scheme, its constraint set remains coherent (i.e. it does not contain any contradiction, see [10,13]) and that *C* is not already in the *R*'s scheme (not even as a redundant constraint); "should be *C*" is the question that *MatBase* asks its users in order to find out whether there is a business rule in the corresponding context stating that *R* must always satisfy constraint *C*.

Consequently, "*R* is, could and should be *C*" means that *MatBase* asks users "*R* should be *C*" only if both the syntactical condition "*R* could be *C*", as well as the semantical one "*R* is *C*" are true.

The order in which *ADDRC* considers the constraint types reflects our experience of more than 45 years in conceptual data modeling: most of the dyadic relationships are asymmetric (as their mathematical counterparts are symmetric), hence irreflexive too, then there are quite a lot of acyclic ones, with all other such constraint types being less usual. For example, in a *Geography* db [13], all 11 dyadic relationships are asymmetric (and hence irreflexive as well).

4. *MatBase* Algorithm for Enforcing the Dyadic Relationship Constraints

Figure 5 presents the Algorithm *AEDRC*, designed for enforcing the dyadic relationship constraints, which is implemented in both *MatBase* versions as well.

```
ALGORITHM AEDRC. Dyadic Relationship Constraints Enforcement
Input: - a db scheme S, its associated RELATIONSHIPS form instance, the user request (check / uncheck), the corresponding dyadic relationship R =
(f → T, g → T) non-relational constraint type c from its current row, and the implied by c set I, as well as the set I' of the constraints implied only by
c and not desired anymore in S;
       - Cancel = False;
Output: if Cancel then S
     else if uncheck then S = S — {c} — I' else S = S ∪ {c} ∪ I;
Strategy:
select user request
 case uncheck:
   if user does not confirm his/her delete request then
     Cancel = True;
     check c's checkbox;          // undo request
   else if c is implied by some subset of constraints C' then
         Cancel = True;
         check c's checkbox;     // undo request
         display "Constraint cannot be deleted as it is implied by C'!";
      else
         loop for all event-driven methods of the classes associated to R
           delete line assigning to Cancel the value returned for c by the
           corresponding constraint type enforcement Boolean function;
         end loop;
         S = S — {c};
         loop for all constraints c' in S that were implied only by c
          if user wishes to keep c' then generate code needed to enforce c';
          else   S = S — {c'};
          uncheck c's checkbox; // remove unwanted implied constraint
          end if;
        end loop;
      end if;
 case check:
   Cancel = isCoherent(c);
   if Cancel then uncheck c's checkbox;     // undo request
           display "Constraint rejected: constraint set would become incoherent!";
   else Cancel = isValid(c);
      if Cancel then uncheck c's checkbox;            // undo request
          display "Constraint cannot be enforced: current db instance violates it!";
      else
       loop for all event-driven methods of the classes associated to R (and generate all those that might be missing)
         inject line assigning to Cancel the value returned for c by the corresponding
         constraint type enforcement Boolean function;
       end loop;
      S = S ∪ {c} ∪ I;
      loop for all constraints c' in I
            check the checkbox corresponding to c';     // store that f also obeys c'
      end loop;
      end if;
   end if;
end select;
End ALGORITHM AEDRC;
```

Figure 5. Algorithm AEDRC (Dyadic Relationship Constraints Enforcement)

DOI: https://doi.org/10.30564/jeisr.v2i2.2090

The Boolean functions *isCoherent* (checking that adding / removing a constraint satisfies or violates coherence) and *isValid* (checking whether the current db instance satisfies a given constraint), as well as many other useful functions for automatically enforcing constraints generated code, are provided by *MatBase* in its *Constraints* library [12,13].

5. Results and Discussion

5.1 Algorithms' Complexity and Optimality

It is very easy to check that both these algorithms are very fast, as they are never infinitely looping and their time complexities are $O(|C|)$ (i.e. linear in the average number of dyadic relationship fundamental constraints, generally between 0 and 3 [13]), for *ADDRC*, and $O(|I|)$ (i.e. linear in the average number of implied constraints by a non-relational dyadic relationship constraint, generally between 0 and 5 [13]), for *AEDRC*, respectively.

ADDRC is trivially not infinitely looping, as each dyadic relationship and corresponding constraint types are considered only once.

In the worse case (in which no non-relational constraint is initially asserted for the current dyadic relationship and none is discovered either), *ADDRC* is asking users all of the 10 possible questions. Coherence and minimality of the constraint sets are almost instantly checked, with only two table row reads [10,13]. Only checking whether the data instance of a dyadic relationship takes time proportional with its cardinal (i.e. the number of rows in the corresponding table). Consequently, it is preferable to assert all such non-relational constraints immediately after defining the relationship schemas, before allowing users to enter data for them.

Moreover, *ADDRC* is also optimal, as, using both math and our decades of conceptual data modeling experience, it asks db designers the minimum number possible of questions for any dyadic relationship.

ADDRC is implemented in the *DoubleClick* event associated to the *MatBase* form *RELATIONSHIPS*.

AEDRC is trivially not infinitely looping either, as any checkbox corresponding to a dyadic relationship constraint type is considered only once.

Moreover, *AEDRC* is optimal too, as

(1) it searches in every object-oriented class only within the event-driven methods and no such method is visited twice and

(2) its implementations merges for deletions both code injections and deletions in a same step, whereas checking of implied constraints by a newly added one is done in internal memory (and are saved in the db together with the one done by users in the current constraint checkbox, when the current row from *RELATIONSHIPS* is saved).

AEDRC is implemented in the *BeforeUpdate / Validating* for checkbox controls associated to the non-relational constraint types of the form *RELATIONSHIPS*.

5.2 Algorithms' Usefulness

The main utility of *ADDRC* is, of course, in the realms of data modeling and db constraints theory, whereas the one of *AEDRC* in the db and db software applications design and development ones: all constraints (business rules) that are governing the sub-universes modelled by dbs, be them relational or not, should be discovered and enforced in the corresponding dbs' schemas; otherwise, their instances might be implausible.

Dyadic relationships have sometimes associated non-relational constraints that may be much more easily discovered by using the assistance algorithm *ADDRC* presented in this paper.

Being transparent to users and automatically generating constraint enforcement code, *AEDRC* not only significantly enhances software architects and developers productivity, but also saves them lot of debugging effort and guarantees a very high quality standard.

6. Conclusion

In summary, we have designed, implemented, and successfully tested in both *MatBase* latest versions (for MS Access and C# and SQL Server) an algorithm for assisting discovery of all non-relational constraints associated to dyadic relationships, another one for enforcing such constraints through automatic code generation, analysed their complexities and optimality, as well as outlined their usefulness for data modelling, db constraints theory, db and db software application design and development practices.

Very many non-relational db constraint types are attached to dyadic relationships (dr). The first algorithm presented in this paper helps users to analyse each such dr exhaustively and intelligently, such that they may discover all non-relational constraints associated to them in the minimum possible time. Moreover, through automatic code generation for enforcing them, *MatBase* significantly increases both software development productivity and quality.

These algorithms are successfully used both in our lectures and labs on Advanced Databases (for the postgraduate students of the Mathematics and Computer Science Department of the Ovidius University, Constanta and the Computer Science Taught in English Department of the Bucharest Polytechnic University) and by two Romanian IT companies developing db software applications for many U.S. and European customers in the Fortune 100 ones.

In fact, *MatBase* is automatically generating code for enforcing non-relational constraints not only for dyadic relationships, but also for object constraints, endofunctions [12,13], the rest of the functions (including Cartesian product ones), as well as for sets [13], which makes it also a formal BRMS and adds (E)MDM to the panoply of tools expressing business rules.

References

[1] Abiteboul, S., Hull, R., Vianu, V. Foundations of Databases. Addison-Wesley, Reading, MA, 1995.

[2] Chen, P. P. The entity-relationship model: Toward a unified view of data. ACM TODS, 1976, 1(1): 9-36.

[3] Codd, E. F. A relational model for large shared data banks. CACM, 1970, 13(6): 377-387.

[4] Gosnell, D., Broecheler, M. The Practitioner's Guide to Graph Data: Applying Graph Thinking and Graph Technologies to Solve Complex Problems. O'Reilly Media, Inc. Sebastopol, CA, 2020.

[5] Jech, T. Set Theory. Springer Monographs in Mathematics (Third Millennium ed.). Springer-Verlag, Berlin, New York, 2003.

[6] Mancas, C. Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume I: The Shortest Advisable Path. Apple Academic Press / CRC Press (Taylor & Francis Group), Waretown, NJ, 2015.

[7] Mancas, C. On the paramount importance of database constraints. J. Inf. Tech. & Soft. Eng., 5(3):1-4. Henderson, NV, 2015.

[8] Mancas, C. Algorithms for key discovery assistance. In: Repa, V., Bruckner, T. (eds). BIR 2016, LNBIP, Springer, Cham, 2016, 261: 322-338.

[9] Mancas, C. On Database Relationships versus Mathematical Relations. Global Journal of Comp. Sci. and Techn.: Soft. and Data Eng., Framingham, MA, 2016, 16(1): 13-16.

[10] Mancas, C. MatBase Constraint Sets Coherence and Minimality Enforcement Algorithms. In: Benczur, A., Thalheim, B., Horvath, T. (eds.), Proc. 22nd ADBIS Conf. on Advances in DB and Inf. Syst., LNCS 11019, Springer, Cham, 2018: 263-277.

[11] Mancas, C. MatBase E-RD Cycles Associated Non-Relational Constraints Discovery Assistance Algorithm. In: Arai, K., Bhatia, R., Kapoor, S. (eds.), Intelligent Computing, Proc. 2019 Computing Conference, AISC Series, Springer, Cham, 2019, 997(1): 390-409.

[12] Mancas, C. Matbase Autofunction Non-Relational Constraints Enforcement Algorithms. IJCSIT, 2019, 11(5): 63-76.

[13] Mancas, C. Conceptual Data Modeling and Database Design: A Completely Algorithmic Approach. Volume II: Refinements for an Expert Path. Apple Academic Press / CRC Press (Taylor & Francis Group), Waretown, NJ, 2021, in press.

[14] Rautenberg, W. A Concise Introduction to Mathematical Logic (3rd ed.). Springer Science+Business Media, NY, 2010.

[15] Ross, R. G. Principles of the Business Rule Approach. Addison-Wesley Professional, Boston, MA, 2003.

[16] Shinavier, J., Wisnesky, R. Algebraic property graphs. Tech. Rep. Cornell Univ., 2019. https://arxiv.org/pdf/1909.04881.pdf

[17] Schultz, P., Wisnesky, R. Algebraic Data Integration. Expanded and corrected version of the paper published in J. Functional Programming, 27, E24, Cambridge Univ. Press, 2017. https://arxiv.org/pdf/1503.03571.pdf

[18] Thalheim, B. Entity-Relationship Modeling: Foundations of Database Technology. Springer-Verlag, Berlin, 2000.

[19] Thom, F., Abdennadher, S. Essentials of Constraint Programming. Springer-Verlag, Berlin, Germany, 2003.