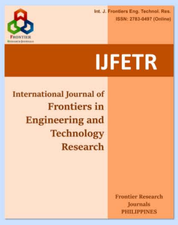
International Journal of Frontiers in
Engineering and Technology Research
Journal homepage: https://frontiersrj.com/journals/ijfetr/
ISSN: 2783-0497 (Online)



(RESEARCH ARTICLE)

On enforcing dyadic-type self-map constraints in *MatBase*

Christian Mancas [*]

Department of Mathematics and Computer Science, Ovidius University at Constanta, Romania.

International Journal of Frontiers in Engineering and Technology Research, 2023, 05(01), 014–026

Publication history: Received on 03 July 2023; revised on 12 August 2023; accepted on 15 August 2023

Article DOI: https://doi.org/10.53294/ijfetr.2023.5.1.0023

Abstract

Self-maps are widely encountered in the sub-universes modeled by databases, from genealogical trees to sports, from education to healthcare, etc. Their properties must be discovered and enforced by the software applications managing such data to guarantee their plausibility. The (Elementary) Mathematical Data Model provides 13 dyadic-type self-map constraint types. *MatBase*, an intelligent data and knowledge base management system prototype, allows database designers to simply declare them by only clicking corresponding checkboxes and automatically generates code for enforcing them. This paper describes the algorithms that *MatBase* uses for enforcing all these 13 self-map constraint types, which may also be used by developers not having access to *MatBase*.

Keywords: Database constraints; Self-maps; Dyadic relations; Modelling as programming; The (Elementary) Mathematical Data Model; *MatBase*

1. Introduction

Very many database (db) sub-universes include self-maps (self-functions, autofunctions, e.g., [1 – 4]). *Self-maps* are cases of both functions (i.e., of type $f : D \to D$) and dyadic relations (i.e., for which the first Canonical projection is the *unity function* of D, $\mathbf{1}_D : D \to D$, $\mathbf{1}_D(x) = x$, $\forall x \in D$, and the second one is f) [5]. As such, besides general function properties (i.e., one-to-oneness / injectivity, ontoness / surjectivity, etc.), they also enjoy most of the dyadic relation type ones.

For example, *Mother* : *PERSONS* → *PERSONS*, *Father* : *PERSONS* → *PERSONS*, *Spouse* : *PERSONS* → *PERSONS*, *KilledBy* : *PERSONS* → *PERSONS*, *RefersTo* : *EMPLOYEES* → *EMPLOYEES*, and *ParentDynasty* : *DYNASTIES* → *DYNASTIES* are self-maps frequently encountered in db schemes.

The (Elementary) Mathematical Data Model ((E)MDM) [2, 4, 6] considers 11 types of dyadic constraints [7, 8]; only the following 8 ones also apply to self-maps: reflexivity, irreflexivity, symmetry, asymmetry, transitivity (idempotency), intransitivity (anti-idempotency), equivalence, and acyclicity. Moreover, canonical surjectivities are also self-maps [2 – 4].

For example, *Mother* and *Father* are irreflexive (i.e., nobody may be his/her mother / father), asymmetric (i.e., nobody may be the parent of his/her parents), anti-idempotent / intransitive (i.e., no parent may be his/her parent), and acyclic (i.e., nobody may be his/her ancestor); as for dynasties similar properties are true, *ParentHouse* too is irreflexive, asymmetric, anti-idempotent, and acyclic; *Spouse* is irreflexive (i.e., nobody may be his/her spouse), symmetric (i.e., whenever w is the spouse of h, h is the spouse of w), and anti-idempotent / intransitive (as it is irreflexive).

[*] Corresponding author: Christian Mancas

On one hand, as with any other constraint (business rule), failing to enforce any of the above ones could lead to storing unplausible data in the corresponding db (e.g., for some persons x and y, *Mother*(x) = x or *Spouse*(x) = y and *Spouse*(y) $\neq x$).

On the other hand, as reflexivity, symmetry, and idempotency constraints are of type tuple generating, enforcing them is also saving time for end-users, as the corresponding self-map values may be automatically generated by the db software applications managing that data.

We recall that [2 – 4]:

- The only reflexive (so equivalence as well) self-maps are the unity functions;
- A self-map is symmetric if and only if (iff) its square is the unity function of its domain;
- A self-map is idempotent iff its square is equal to itself;
- A self-map is a canonical surjection iff it is totally defined, onto, and idempotent.

Please note that, as for any other db interesting functions, self-maps may be partially defined: e.g., there are persons with unknown parents, persons not married, dynasties not having parent ones, etc. For such cases, db theory considers a countable distinguished set of *null values* denoted NULLS. Hence, generally, a self-map definition might be of the type $f : D \to D \cup$ NULLS.

Consequently, apart from the 9 "standard" self-map dyadic-type above mentioned constraints, the (E)MDM also considers the following 4 additional ones: null-reflexivity, null-symmetry, null-idempotency, and null-equivalence [2 – 4]. $f : D \to D \cup$ NULLS is said to be:

- *null-reflexive*, iff $f(x) \in \{x\} \cup$ NULLS, $\forall\, x \in$D;
- *null-symmetric*, iff $f(x) = y \Rightarrow f(y) \in \{ x \} \cup$ NULLS, $\forall\, x, y \in$D;
- *null-idempotent*, iff $f(x) = y \Rightarrow f(y) \in \{ y \} \cup$ NULLS, $\forall x, y \in$D;
- *null-equivalence*, iff it is null-reflexive.

Please note that, in dbs, reflexive atomic self-maps are of no use: why would anybody duplicate the surrogate primary key of a table? However, reflexivity of composed self-maps is crucial in correctly modeling some sub-universes of discourse. For example, given functions *LDrive* : *FOLDERS* → *LDRIVES* (storing for each file folder the logic drive on which it is stored) and *RootFolder* : *LDRIVES* → *FOLDERS* (storing for each logic drive its corresponding root file folder), the composed function *LDrive* ° *RootFolder* must be reflexive, i.e. *LDrive*(*RootFolder*(x)) = x, for any x in *LDRIVES*, as the root folder of any logic drive must belong to that logic drive.

Of course, the dyadic-type self-map types of constraints are not enough for guaranteeing data plausibility, not even for *PERSONS*: as usual, all other existing constraints in the corresponding sub-universe should also be enforced. For example, *Mother* • *Spouse acyclic* (No woman may be the spouse of one of her children, grandchildren, or descendants.), *Father* • *Spouse acyclic* (No man may be the spouse of one of his children, grandchildren, or descendants.), etc.

Unfortunately, while, for example, uniqueness may be enforced by almost any commercial Database Management System (DBMS), with unique indexes, no such DBMS may enforce dyadic-type self-map constraints. Consequently, developers must enforce them into the software applications that manage corresponding dbs (through either extended SQL triggers or event-driven methods of high-level programming languages embedding SQL).

MatBase [2, 3, 4, 7, 8, 10] is an intelligent prototype data and knowledge base management system, based on both the (E)MDM, the Entity-Relationship (E-R) Data Model (E-RDM) [1, 12, 13], the Relational Data Model (RDM) [1, 14, 15], and Datalog [4, 15, 16].

Fortunately, *MatBase* provides, through its (E)MDM interface, both a very user-friendly experience to db architects (e.g., for the *Mother*, *Father*, *Mother* • *Spouse*, and *Father* • *Spouse* of *PERSONS* above, you only need to click their corresponding *Acyclic* checkboxes, as their other 3 properties are implied by it, so redundant) and its associated code-generating power, which is both constructing underlying db tables, standard MS Windows forms for them, as well as event-driven code in their classes for enforcing the corresponding constraints that cannot be enforced by the available DBMS.

As such, *MatBase* is not only saving developing time, but also saves testing and debugging time, which promotes the 5th programming generation – modelling as programming [9 – 11]. This paper presents the pseudocode algorithms used by *MatBase* to automatically generate code for enforcing dyadic-type self-map constraints, being a continuation of [3], which was mainly focused on assisting the process of detecting self-map constraints: it refines its *A2 Algorithm* (which is a very high level one, mainly dealing with the coherence and minimality of the sets of self-map constraints) for each type of dyadic-type self-map constraints.

Other approaches related to the (E)MDM are based on business rules management (BRM) [17 – 22] and their corresponding implemented systems (BRMS) and process managers (BPM), like the IBM Operational Decision Manager [23], IBM Business Process Manager [24], Red Hat Decision Manager [25], Agiloft Custom Workflow/BPM [26], etc.

They are generally based on XML (but also on the Z notation, the Business Process Execution Language, the Business Process Modeling Notation, the Decision Model and Notation models, Drools Rule Language files, guided decision tables, or the Semantics of Business Vocabulary and Business Rules).

This is the only other field of endeavor trying to systematically deal with business rules, even if informally. However, this is not done at the db design level, but at the software application one, and without providing automatic code generation.

From this perspective, (E)MDM also belongs to the panoply of tools expressing business rules and *MatBase* is also a BRMS, but a formal, automatically code generating one.

2. Material and Methods

Let $f : D \to D \cup \text{NULLS}$ be an arbitrary self-map defined on and taking values from a set D. For enforcing self-map type constraints on f, D must have a Graphic User Interface (GUI) form (associated to its corresponding table) whose class D must contain the following private variables and event-driven methods (see Figure 1):

- Definition of an integer variable *fOldValue*, as well as of Boolean ones *fSymmForce1*, *fSymmForce2*, and *fIdemForce1*;
- A *Current*(x) method, to be automatically called each time the cursor of the D's form enters a new element (line, row, record) x of its underlying data table;
- A *f_BeforeUpdate*(x) method, to be automatically called each time a new or existing element x of its underlying data whose *fOldValue* value for column f was modified and is about to be saved in the db;
- A *BeforeUpdate*(x) method, to be automatically called each time a new or existing element x of its underlying data whose values for any column was modified is about to be saved in the db;
- An *AfterUpdate*(x) method, to be automatically called each time the new values of an existing element x of its underlying data were successfully just saved to the db.

All these methods and variables are automatically generated by *MatBase* the first time it needs them. Code insertion in all these methods is always done immediately before their blank code line (see Figure 1).

Moreover, *MatBase* has in its public *General* library the function definition shown in Figure 2, needed for enforcing self-map acyclicities.

2.1. Enforcing reflexivity and null-reflexivity constraints

According to the reflexivity definition, enforcing such constraints for f requires that:

- Each time a new element x is added to D, $f(x)$ must be automatically set to x if f is totally defined.
- Modification of $f(x)$ must be made only to x or a null value (if f is not totally defined).
- Deletion of $f(x)$ may be possible only if f is not totally defined.

Consequently, *MatBase* adds the pseudocode algorithms from Figure 3 to the corresponding methods from Figure 1.

2.2. Enforcing irreflexivity constraints

According to the irreflexivity definition, enforcing such constraints for *f* requires rejecting attempts to save for *f*(*x*) the value *x*. Consequently, *MatBase* adds the pseudocode algorithm from Figure 4 to the method *f_BeforeUpdate* from Figure 1.

2.3. Enforcing symmetry and null-symmetry constraints

According to the symmetry definition, enforcing such constraints for *f* requires that modifications of *f*(*x*) must be made only to *x*, an existing *y* from *D*, or a null value (if *f* is not totally defined) and:

```
int fOldValue;
Boole fsymmForce1, fsymmForce2;
Boole fidemForce1;
Method Current(x)
fOldValue = f(x);
fsymmForce1 = False;
fsymmForce2 = False;
fidemForce1 = False;

end method;
Method f_BeforeUpdate(x)
Boole Cancel = False;

if Cancel then restore fOldValue;
           deny saving f(x) to the db;
end if;
end method;
```

```
Method BeforeUpdate(x)
 Boole Cancel = False;

if Cancel then
   deny saving x's new data to the db;
end method;
Method AfterUpdate(x)
if fsymmForce1 then f(f(x)) = x;
   fsymmForce1 = False;
end if;
if fsymmForce2 then f(fOldValue) = null;
   fsymmForce2 = False;
end if;
if fidemForce1 then f(f(x)) = f(x);
   fidemForce1 = False;
end if;
end method;
```

Figure 1 The initial pseudocode automatically generated by *MatBase* for the *D*'s class

```
public Boolean function IsPath(text D, int x, text f, int y)
// returns true if there is a path in the f : D → D's graph from f(x)=y to x, false otherwise
int v;
IsPath = false;                        // no path found yet
if y ∈ NULLS then                      // no path possible when y ∈ NULLS
  if x == y then IsPath = true;        // eliminate cycles of length 1
    display "Request rejected: acyclic self-maps are irreflexive!";
  else
   if f(y) == x then IsPath = true;    // eliminate cycles of length 2
     display "Request rejected: acyclic self-maps are asymmetric!";
   else                                // look for cycles of length at least 3  v = y;
     while v ∉ NULLS and not IsPath
       v = f(v);
       if v ∉ NULLS then
         if v == x then IsPath = true;  display "Request rejected:        // path found!
                                 f is acyclic and f(x) = y would create a cycle in its graph!";
         end if;                       // of if v == x
       end if;                         // of if v ∉ NULLS
     end while;
   end if;                             // of if f(y) == x
  end if;                              // of if x == y
end if;                                // of if y ∈ NULLS
end Function IsPath;
```

Figure 2 The pseudocode of function *IsPath* from the *MatBase*'s *General* library

Code added to method *f_BeforeUpdate*:

```
// f reflexive
if not Cancel then
    if f is totally defined and $f(x) \in$ NULLS then Cancel = True;
    else if $f(x) \notin$ NULLS and $f(x) \neq x$ then Cancel = True;
     if Cancel then display "Request rejected: as f is reflexive, $f(x)$ should be equal to $x$!";
end if;
```

Code added to method *BeforeUpdate*:

```
// f reflexive
if not Cancel then
    if f is totally defined and $f(x) \in$ NULLS then Cancel = True; display "Request rejected:
                        as f is totally defined, please chose for it a value from D's elements!";
  end if;
end if;
```

Figure 3 Code added to the methods from Figure 1 when *f* is reflexive

```
// f irreflexive
if not Cancel then
    if $f(x) == x$ then Cancel = True; display "Request rejected: as f is irreflexive, $f(x) \neq x$!";
end if;
```

Figure 4 Code added to method *f_BeforeUpdate* from Figure 1 when *f* is irreflexive

- New *f*'s value may be null only if *f* is not total;
- If old $f(x)$'s value was null and its new value is $y \neq x$, $y \in D$, then:
  - Modification must be rejected if *f* is reflexive as well;
  - If *f* is totally defined, modification must be rejected if $f(y) \neq y$ (as *y* is then symmetrically paired with some *z*), whereas otherwise $f(y)$ must be set to *x*;
  - Else (i.e., *f* accepts nulls) if $z = f(y)$ is not null then $f(z)$ must be set to null and, anyhow, $f(y)$ must be set to *x*;
- If old $f(x)$'s value was a not null *z* and its new value is $y \neq x$, $y \notin D$, then:
  - Modification must be rejected if $z = x$ and *f* is reflexive as well;
  - If *f* is totally defined, modification must be rejected if $f(y) \neq y$ (as *y* is then symmetrically paired with some *u*, just as *z* is paired with *x*), whereas otherwise $f(y)$ must be set to *x*;
  - Else (i.e. *f* accepts nulls) if $u = f(y)$ is not null then $f(u)$ must be set to null and, anyhow, $f(y)$ must be set to *x*;
- If old $f(x)$'s value was a not null $z \neq x$ and its new value is *x*, then:
  - If *f* is irreflexive then modification must be rejected;
  - Else $f(z)$ must be set to a null value.

Consequently, *MatBase* adds the pseudocode algorithm from Figure 5 to the method *f_BeforeUpdate* from Figure 1.

2.4. Enforcing asymmetry constraints

According to the asymmetry definition, enforcing such constraints for *f* requires that each time a new not null value $y \in D$, $y \neq x$, is about to be saved for $f(x)$, saving must be canceled if $f(y) = x$. Consequently, *MatBase* adds the pseudocode algorithm from Figure 6 to the method *f_BeforeUpdate* from Figure 1.

```
// $f$ symmetric
if not Cancel then
   if $f$ is totally defined and $f(x) \in$ NULLS then Cancel = True;
   else
      if fOldValue $\in$ NULLS then
         if $f(x) \neq x$ then
            if $f$ reflexive then Cancel = True;
            else if $f$ is totally defined then Cancel = True;
                  else fsymmForce1 = True;
                        if $f$(fOldValue) $\notin$ NULLS then fsymmForce2 = True;
                  end if;                         // of if $f$ is totally defined
            end if;                               // of if $f$ reflexive
         end if;                                  // of if $f(x) \neq x$
      else                                        // of fOldValue $\in$ NULLS
         if fOldValue $\neq x$ and $f(x) == x$ then
               if $f$ is irreflexive then Cancel = True;
               else fsymmForce2 = True;
         end if;                                  // of if fOldValue $\neq x$ and $f(x) == x$
      end if;                                     // of if fOldValue $\in$ NULLS
   end if;                                        // of if $f$ is totally defined and $f(x) \in$ NULLS
   if Cancel then display “Request rejected: $f$ is symmetric!”;
end if;
```

Figure 5 Code added to method *f_BeforeUpdate* from Figure 1 when *f* is symmetric

```
// $f$ asymmetric
if not Cancel and $f(x) \neq x$ and $f(x) \notin$ NULLS and $f(f(x)) == x$ then Cancel = True;
          display “Request rejected: $R$ is asymmetric!”;
end if;
```

Figure 6 Code added to method *f_BeforeUpdate* from Figure 1 when *f* is asymmetric

2.5. Enforcing idempotency and null-idempotency constraints

According to the idempotency definition, enforcing such constraints for *f* requires that modifications of *f*(*x*) must be made only to *x*, an existing *y* from *D*, or a null value (if *f* is not totally defined) and:

- New *f*'s value may be null only if *f* is not total;
- If old *f*(*x*)'s value was null and its new value is $y \neq x, y \in D$, then:
  - ○ Modification must be rejected if *f* is irreflexive as well;
  - ○ Otherwise *f*(*y*) must be set to *y*;
- If old *f*(*x*)'s value was a not null *z* and its new value is $y \neq x, y \in D$, then:
  - ○ Modification must be rejected if $z = x$ and *f* is reflexive as well;
  - ○ If *f* is totally defined, modification must be rejected if $f(y) \neq y$, whereas otherwise *f*(*y*) must be set to *y*;
  - ○ Else (i.e. *f* accepts nulls) if $u = f(y)$ is not null then *f*(*u*) must be set to null and, anyhow, *f*(*y*) must be set to *y*;
- if old *f*(*x*)'s value was a not null $z \neq x$ and its new value is *x*, then:

- If *f* is irreflexive then modification must be rejected;
- Else if *f*(*z*) is not equal to *z*, then it must be set to a null value, if *f* is not totally defined, and otherwise the modification must be rejected.

Consequently, *MatBase* adds the pseudocode algorithm from Figure 7 to the method *f_BeforeUpdate* from Figure 1.

```
// f idempotent
if not Cancel then
  if f is totally defined and $f(x) \in$ NULLS then Cancel = True;
  else
    if fOldValue $\in$ NULLS then
      if $f(x) \neq x$ then
        if f irreflexive then Cancel = True;
        else fidemForce1 = True;
      end if;                              // of if $f(x) \neq x$
    else                                   // of if fOldValue $\in$ NULLS
      if fOldValue $\neq x$ and $f(x) == x$ then
        if f is irreflexive then Cancel = True;
        else if f(fOldValue) $\neq$ fOldValue then
              if f is totally defined then Cancel = True;
              else fsymmForce2 = True;
            end if;                        // of if f(fOldValue) $\neq$ fOldValue
        end if;                            // of if f is irreflexive
      end if;                              // of if fOldValue $\neq x$ and $f(x) == x$
    end if;                                // of if fOldValue $\in$ NULLS
  end if;                                  // of if f is totally defined and $f(x) \in$ NULLS
  if Cancel then display "Request rejected: f is idempotent!";
end if;
```

Figure 7 Code added to method *f_BeforeUpdate* from Figure 1 when *f* is idempotent

2.6. Enforcing anti-idempotency constraints

According to the anti-idempotency definition, enforcing such constraints for *f* requires rejecting attempts to save for *f*(*x*) a not null value *y* with *f*(*y*) = *y*. Consequently, *MatBase* adds the pseudocode algorithm from Figure 8 to the method *f_BeforeUpdate* from Figure 1.

```
// f anti-idempotent
if not Cancel then if $f(x) \notin$ NULLS and $f(f(x)) == f(x)$ then Cancel = True;
                        display "Request rejected: f anti-idempotent!";
                   end if;
end if;
```

Figure 8 Code added to method *f_BeforeUpdate* from Figure 1 when *f* is anti-idempotent

2.7. Enforcing equivalence and null-equivalence constraints

According to the definition of relation equivalence, enforcing it for *f* requires that *f* be both reflexive, symmetric, and idempotent. However, as the only reflexive self-maps are the unity ones, symmetry and idempotency are redundant in this case. Consequently, equivalence and null-equivalence are enforced by the algorithms from subsection 2.1 (Figure 3).

2.8. Enforcing acyclicity constraints

According to the acyclicity definition, enforcing such constraints for f requires rejecting any not null value $f(x)$ whenever there is a path in f's graph from it to x, i.e., there exists a set of elements $\{x_1, …, x_n\} \subseteq D$, $n \geq 0$, such that $f(x) = x_1, f(x_1) = x_2, …, f(x_n) = x$.

Consequently, *MatBase* adds the pseudocode algorithm from Figure 9 to the method *f_BeforeUpdate* from Figure 1.

```
// f acyclic
if not Cancel and $f(x) \notin$ NULLS then Cancel = IsPath("D", x, "f", f(x));
if Cancel then display "Request rejected: f acyclic!";
```

Figure 9 Code added to method *f_BeforeUpdate* from Figure 1 when *f* is acyclic

2.9. Enforcing canonical surjectivity constraints

According to the canonical surjectivity characterization theorem, enforcing such constraints for f requires to enforce f's both totality, ontoness, and idempotency. Please recall that *ontoness* (*surjectivity*) requires that, for any element y of D, there is at least one element x of D such that $y = f(x)$.

Consequently, *MatBase* adds the pseudocode algorithms from Figures 7 and 10 to the corresponding methods from Figure 1.

Code added to method *f_BeforeUpdate*:

```
// f canonical surjection
if not Cancel then
   if f(x) $\in$ NULLS then Cancel = True;
   else if f(x) $\notin$ D then Cancel = True;
   if Cancel then display "Request rejected: as f is a
                  canonical surjection, please specify a
                  value for it from D's elements!"
end if;
```

Code added to method *BeforeUpdate*:

```
// f canonical surjection
if not Cancel then
   if f(x) $\in$ NULLS then Cancel = True;
   if Cancel then display "Request rejected: as f is a
                  canonical surjection, please specify a
                  value for it from D's elements!"
end if;
```

Figure 10 Code added to the corresponding methods from Figure 1 when *f* is a canonical surjection

2.10. The *MatBase* algorithm for enforcing above constraint types

Figures 11 to 14 show the *MatBase* Algorithm *A9DSM* for enforcing self-map constraints.

3. Result and Discussion

For example, it is straightforward to check that applying the Algorithm *A9DSM* from Figure 11 to *D* = *PERSONS* and its self-maps *Mother*, *Father*, and *Spouse* from section 1, *MatBase* automatically generates for *D's* class the pseudocode shown in Figure 15.

Generally, the Algorithm *A9DSM* from Figure 11 automatically generates code that is guaranteeing data plausibility for any self-map for which all its properties are declared to *MatBase* as corresponding constraints, while also automatically generating appropriate data values for reflexivities, symmetries, and idempotencies, thus saving most of the developing, testing, and data entering effort.

Moreover, please note that the (E)MDM also includes constraints on binary homogeneous function products [2,4,5] like *Mother* • *Spouse*, *Father* • *Spouse*, *Mother* • *Father* (which is irreflexive, asymmetric, inEuclidian, and acyclic), all of them defined on *PERSONS* and taking values from *PERSONS*^{2} etc., which are particular cases of dyadic relationships as well.

Future research will be devoted to describing the code that *MatBase* automatically generates for enforcing the constraints associated with homogeneous binary function products.

```
MatBase Algorithm A9DSM for enforcing dyadic-type self-map constraints

Input: A db software application SA over a set D, a self-map f : D → D, and a constraint c of subtype s on f
Output: SA augmented such as to enforce c as well

Strategy:
switch (s)
  case s: reflexivity, equivalence
        enforceReflexivity;
        break;
  case s: irreflexivity
        add to method f_BeforeUpdate from Figure 1 the code from Figure 4;
        break;
  case s: symmetry
        enforceSymmetry;
        break;
  case s: asymmetry
        add to method f_BeforeUpdate from Figure 1 the code from Figure 6;
        break;
  case s: idempotency
        enforceIdempotency;
        break;
  case s: anti-idempotency
        add to method f_BeforeUpdate from Figure 1 the code from Figure 8;
        break;
  case s: acyclicity
        add to method f_BeforeUpdate from Figure 1 the code from Figure 9;
        break;
  case s: canonical surjectivity
        add to the corresponding methods from Figure 1 the code from Figure 10;
        if f is not declared as idempotent then enforceIdempotency;
        break;
end switch;
End MatBase Algorithm A9SM for enforcing self-map constraint
```

Figure 11 *MatBase* algorithm *A9DSM* for enforcing dyadic-type self-map constraints

```
Method enforceReflexivity

add to methods f_BeforeUpdate and BeforeUpdate from Figure 1 the code from Figure 3;
```

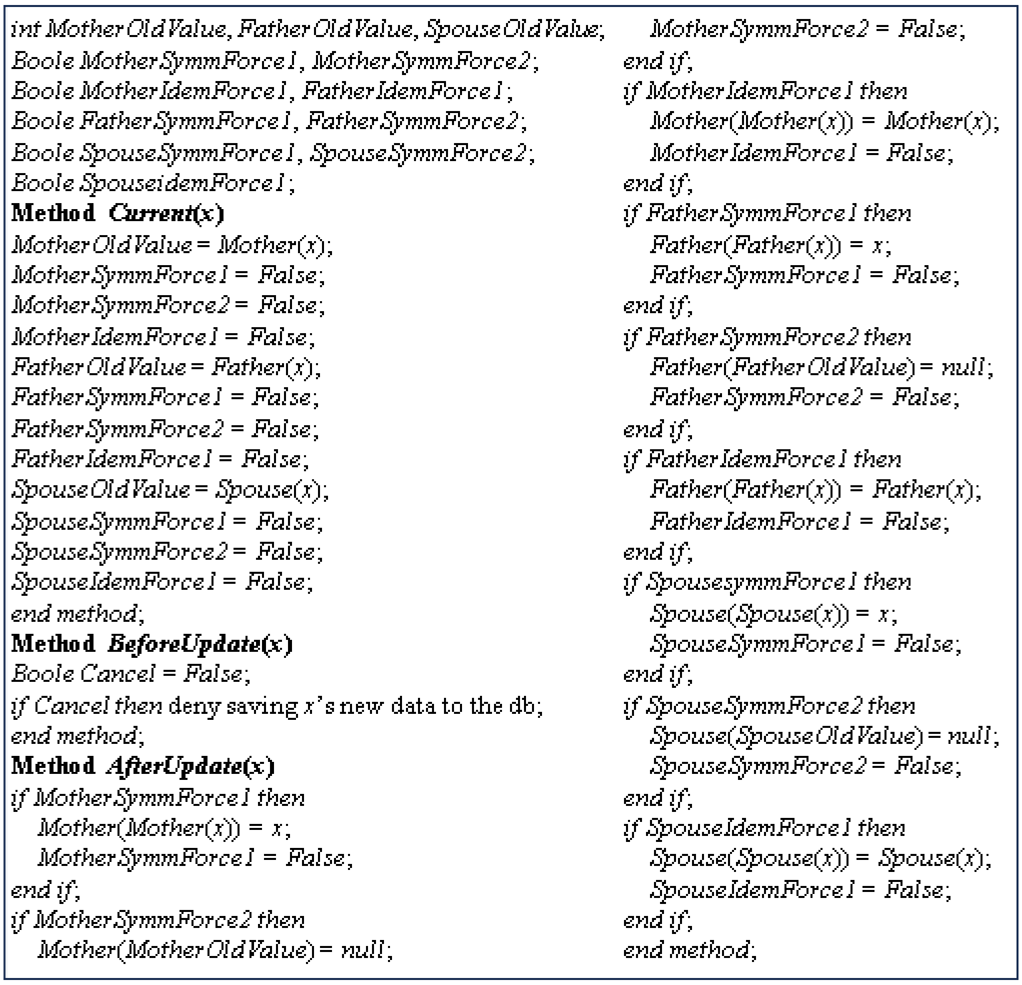
Figure 12 Method *enforceReflexivity* of Algorithm *A9DSM*

```
Method enforceSymmetry

add to method f_BeforeUpdate from Figure 1 the code from Figure 5;
```

Figure 13 Method *enforceSymmetry* of Algorithm *A9DSM*

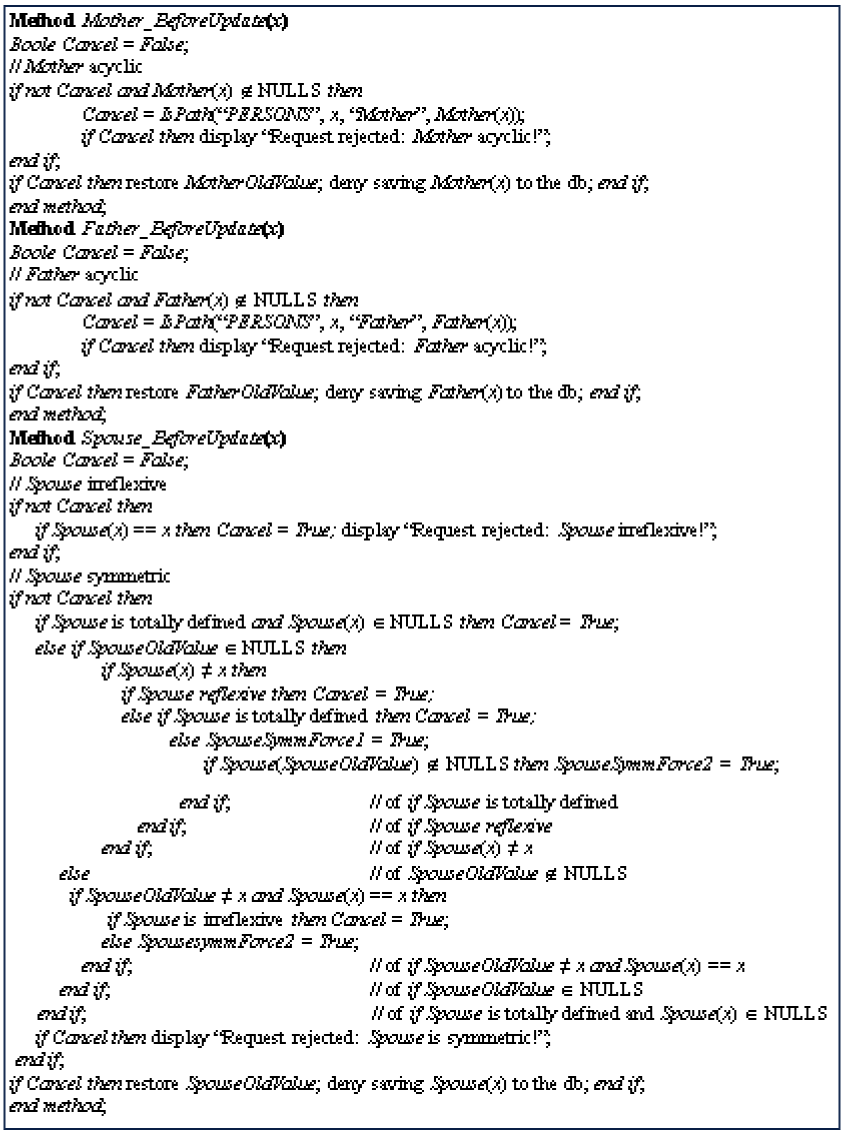
```
Method enforceIdempotency

add to method f_BeforeUpdate from Figure 1 the code from Figure 7;
```

Figure 14 Method *enforceIdempotency* of Algorithm *A9DSM*



```
int MotherOldValue, FatherOldValue, SpouseOldValue;
Boole MotherSymmForce1, MotherSymmForce2;
Boole MotherIdemForce1, FatherIdemForce1;
Boole FatherSymmForce1, FatherSymmForce2;
Boole SpouseSymmForce1, SpouseSymmForce2;
Boole SpouseidemForce1;
Method Current(x)
MotherOldValue = Mother(x);
MotherSymmForce1 = False;
MotherSymmForce2 = False;
MotherIdemForce1 = False;
FatherOldValue = Father(x);
FatherSymmForce1 = False;
FatherSymmForce2 = False;
FatherIdemForce1 = False;
SpouseOldValue = Spouse(x);
SpouseSymmForce1 = False;
SpouseSymmForce2 = False;
SpouseIdemForce1 = False;
end method;
Method BeforeUpdate(x)
Boole Cancel = False;
if Cancel then deny saving x's new data to the db;
end method;
Method AfterUpdate(x)
if MotherSymmForce1 then
   Mother(Mother(x)) = x;
   MotherSymmForce1 = False;
end if;
if MotherSymmForce2 then
   Mother(MotherOldValue) = null;
   MotherSymmForce2 = False;
end if;
if MotherIdemForce1 then
   Mother(Mother(x)) = Mother(x);
   MotherIdemForce1 = False;
end if;
if FatherSymmForce1 then
   Father(Father(x)) = x;
   FatherSymmForce1 = False;
end if;
if FatherSymmForce2 then
   Father(FatherOldValue) = null;
   FatherSymmForce2 = False;
end if;
if FatherIdemForce1 then
   Father(Father(x)) = Father(x);
   FatherIdemForce1 = False;
end if;
if SpousesymmForce1 then
   Spouse(Spouse(x)) = x;
   SpouseSymmForce1 = False;
end if;
if SpouseSymmForce2 then
   Spouse(SpouseOldValue) = null;
   SpouseSymmForce2 = False;
end if;
if SpouseIdemForce1 then
   Spouse(Spouse(x)) = Spouse(x);
   SpouseIdemForce1 = False;
end if;
end method;
```

Figure 15 *MatBase* automatically generated code in class *PERSONS* for enforcing dyadic-type constraints on self-maps *Mother*, *Father*, and *Spouse*



```
Method Mother_BeforeUpdate(x)
Boole Cancel = False;
// Mother acyclic
if not Cancel and Mother(x) ∉ NULLS then
            Cancel = IsPath("PERSONS", x, "Mother", Mother(x));
            if Cancel then display "Request rejected: Mother acyclic!";
end if;
if Cancel then restore MotherOldValue; deny saving Mother(x) to the db; end if;
end method;
Method Father_BeforeUpdate(x)
Boole Cancel = False;
// Father acyclic
if not Cancel and Father(x) ∉ NULLS then
            Cancel = IsPath("PERSONS", x, "Father", Father(x));
            if Cancel then display "Request rejected: Father acyclic!";
end if;
if Cancel then restore FatherOldValue; deny saving Father(x) to the db; end if;
end method;
Method Spouse_BeforeUpdate(x)
Boole Cancel = False;
// Spouse irreflexive
if not Cancel then
    if Spouse(x) == x then Cancel = True; display "Request rejected: Spouse irreflexive!";
end if;
// Spouse symmetric
if not Cancel then
    if Spouse is totally defined and Spouse(x) ∈ NULLS then Cancel = True;
    else if SpouseOldValue ∈ NULLS then
            if Spouse(x) ≠ x then
               if Spouse reflexive then Cancel = True;
               else if Spouse is totally defined then Cancel = True;
                    else SpouseSymmForce1 = True;
                       if Spouse(SpouseOldValue) ∉ NULLS then SpouseSymmForce2 = True;

                    end if;                 // of if Spouse is totally defined
                 end if;                    // of if Spouse reflexive
            end if;                         // of if Spouse(x) ≠ x
       else                                 // of SpouseOldValue ∉ NULLS
        if SpouseOldValue ≠ x and Spouse(x) == x then
             if Spouse is irreflexive then Cancel = True;
            else SpousesymmForce2 = True;
          end if;                           // of if SpouseOldValue ≠ x and Spouse(x) == x
       end if;                              // of if SpouseOldValue ∈ NULLS
    end if;                                 // of if Spouse is totally defined and Spouse(x) ∈ NULLS
    if Cancel then display "Request rejected: Spouse is symmetric!";
 end if;
if Cancel then restore SpouseOldValue; deny saving Spouse(x) to the db; end if;
end method;
```

Figure 15 (Continued)

4. Conclusion

Not enforcing any existing business rule from the sub-universe managed by a db software application allows saving unplausible data in its db. This paper presents the algorithms needed to enforce the dyadic-type self-map constraint types from the (E)MDM, which are implemented in *MatBase*, an intelligent DBMS prototype. Moreover, as it automatically generates the corresponding code, *MatBase* is a tool of the 5th generation programming languages – *modelling as programming*: db and software architects only need to assert the properties of the self-maps (and not only, but of all other (E)MDM constraint types), while *MatBase* saves the corresponding developing, testing, and debugging time. Obviously, these algorithms may also be used by developers not having access to *MatBase*.

Compliance with ethical standards

Acknowledgments

This research was not sponsored by anybody and nobody other than its author contributed to it.

References

[1] Mancas C. Conceptual data modeling and database design: A completely algorithmic approach. Volume I: The shortest advisable path. Palm Bay: Apple Academic Press / CRC Press (Taylor & Francis Group); 2015.

[2] 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. Cham, Switzerland: Springer; 2018; p. 263–277.

[3] Mancas C. Matbase Autofunction Non-relational Constraints Enforcement Algorithms. International Journal of Computer Science & Information Technology. 2019 Oct, 11(5):63–76.

[4] Mancas C. Conceptual data modeling and database design: A completely algorithmic approach. Volume II: Refinements for an expert path. Palm Bay: Apple Academic Press / CRC Press (Taylor & Francis Group); in press.

[5] O'Leary ML. A first course in mathematical logic and set theory. Hoboken: John Wiley & Sons; 2016.

[6] Mancas C. On Knowledge Representation Using an Elementary Mathematical Data Model. In: Hamza M, ed. Proc. 1st IASTED Int. Conf. on Inf. and Knowledge Sharing (IKS 2002). Calgary, Canada: ACTA Press, 2002; p. 206–211.

[7] Mancas C. On Detecting and Enforcing the Non-Relational Constraints Associated to Dyadic Relations in MatBase. Journal of Electronic & Information Systems. 2020 Oct, 2(2):1-8.

[8] Mancas C. On Enforcing Dyadic Relationship Constraints in MatBase. World Journal of Advanced Engineering Technology & Sciences. 2023 Jul, 2(2):1-12.

[9] Thalheim B. The Future: Modelling as Programming. Model-based development, modelling as programming case studies [internet]. Kiel University, Germany, © 2020 [cited 2023 Jul 27]. Available from: https://www.youtube.com/watch?v=tww7LuVzYco&feature=youtu.be.

[10] Mancas C. MatBase – a Tool for Transparent Programming while Modeling Data at Conceptual Levels. In: Natarajan M. et al, eds. Proc. 5th Int. Conf. on Comp. Sci. & Inf. Techn. (CSITEC 2019). Chennai, India: AIRCC Pub. Corp.; 2019; p. 15–27.

[11] Mancas C. On Modelware as the 5th Generation of Programming Languages. Acta Scientific Computer Sciences. 2020 Sep, 2(9):24–26.

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

[13] Thalheim B. Entity-Relationship Modeling – Foundations of Database Technology. Berlin, Germany: Springer-Verlag; 2000.

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

[15] Abiteboul S., Hull R., Vianu V. Foundations of Databases. Reading, MA: AddisonWesley; 1995.

[16] Maier D., Warren D. S. Computing with Logic: Logic Programming with Prolog. Menlo Park, CA: Benjamin/Cummings; 1988.

[17] Halle von B. Business Rules Applied: Building Better Systems Using the Business Rules Approach. New-York, NY: John Wiley & Sons; 2001.

[18] Morgan T. Business Rules and Information Systems: Aligning IT with Business Goals. Boston, MA: Addison-Wesley Professional; 2002.

[19] Weiden M., Hermans L., Schreiber G., van der Zee S. (2002). Classification and Representation of Business Rules. In: Proc. 2002 European Bus. Rules Conf. https://www.researchgate.net/publication/251521215_Classification_and_Representation_of_ Business_Rules.

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

[21] Halle von B., Goldberg L. The Business Rule Revolution. Running Businesses the Right Way. Cupertino, CA: Happy About; 2006.

[22] Taylor J. Decision Management Systems: A Practical Guide to Using Business Rules and Predictive Analytics. Indianapolis, IN: IBM Press; 2019.

[23] Chen W.-J. et al. Systems of Insights for Digital Transformation. Using IBM Operational Decision Manager Advanced and Predictive Analytics [internet]. Armonk, NY: ibm.com/redbooks © 2015 [cited 2023 Jul 27]. Available from: https://www.redbooks.ibm.com/redbooks/pdfs/sg248293.pdf.

[24] Dyer L. et al. Scaling BPM Adoption from Project to Program with IBM Business Process Manager, 2nd ed. [internet]. Armonk, NY: ibm.com/redbooks. © 2012 [cited 2023 Jul 27]. Available from: http://www.redbooks.ibm.com/redbooks/pdfs/sg247973.pdf.

[25] Red Hat Developer. Red Hat Decision Manager [internet]. Raleigh, NC: Red Hat, Inc. © 2023 [cited 2023 Jul 27]. Available from: https://developers.redhat.com/products/red-hatdecision-manager/download.

[26] Agiloft Inc. Standard System Documentation [internet]. Redwood City, CA: Agiloft Inc. © 2018 [cited 2023 Jul 27]. Available from: https://wiki.agiloft.com/display/SD/Documentation+Archive?preview=/31199110/43450819/ standard-kb-documentation.pdf.