On enforcing dyadic relationship constraints in MatBase

Christian Mancas *

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

World Journal of Advanced Engineering Technology and Sciences, 2023, 09(02), 298–311

Publication history: Received on 06 June 2023; revised on 18 July 2023; accepted on 21 July 2023

Article DOI: https://doi.org/10.30574/wjaets.2023.9.2.0211

Abstract

Dyadic relationships 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, in order to guarantee their plausibility. The (Elementary) Mathematical Data Model provides 11 dyadic relationship 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 11 dyadic relationship constraint types.

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

1. Introduction

Very many database (db) sub-universes include dyadic relationships (e.g., Lawvere and Rosebrugh 2003; Mancas 2023). For example, let us consider soccer championships ones, where, besides a set of TEAMS and one of CITIES, in order to store the results a MATCHES set is needed as well. MATCHES is a dyadic relationship over TEAMS, i.e., a subset of the Cartesian product TEAMS × TEAMS, with, e.g., its first canonical projection denoted Host : MATCHES → TEAMS and its second one denoted Visitor : MATCHES → TEAMS.

Dyadic relationships have very interesting properties, among which the (Elementary) Mathematical Data Model ((E)MDM, Mancas 2002, 2018, 2023) considers the following 11 ones: connectivity, reflexivity, irreflexivity, symmetry, asymmetry, transitivity, intransitivity, Euclideanity, inEuclideanity, equivalence, and acyclicity.

For example, MATCHES is connected (i.e., in any championship, any team should play against all other teams), irreflexive (i.e., no team ever plays against itself), symmetric (i.e., in any championship, for any match <host, visitor> there should also be a match <visitor, host>), transitive (i.e., in any championship, for any matches between teams <t₁, t₂> and <t₂, t₃> there should also be a match <t₁, t₃>), and Euclidean (i.e., in any championship, for any matches between teams <t₁, t₂> and <t₁, t₃> there should also be a match <t₂, t₃>).

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., matches <Chelsea, Chelsea> or missing matches).

On the other hand, as except for the irreflexivity, all other above constraints on MATCHES are of type tuple generating, enforcing them is also saving time for end-users, as they only have to enter data for CITIES and TEAMS, while the system is automatically generating corresponding MATCHES <Host, Visitor> data pairs, with end-users then only having to enter calendar dates and scores for matches.
Of course, the dyadic relationship constraints are not enough for guaranteeing data plausibility, not even for this simple db example: as usual, all other existing constraints in the corresponding sub-universe should also be enforced. For example, names of cities and teams should be unique (i.e., both Team: TEAMS \rightarrow \text{ASCII}(32) and City: CITIES \rightarrow \text{ASCII}(32) must be one-to-one), no team may play more than one match a day (i.e., both Host \bullet MatchDate : MATCHES \rightarrow TEAMS \times [13-Aug-2021, 22-May-2022] and Visitor \bullet MatchDate : MATCHES \rightarrow TEAMS \times [13-Aug-2021, 22-May-2022] must be minimally one-to-one), 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 relationship 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).

Fortunately, our MatBase intelligent system provides, through its (E)MDM interface, both a very user-friendly experience to db architects (e.g., for MATCHES above, you only need to click its corresponding Connected, Irreflexive, Symmetric, Transitive, and Euclidean checkboxes) 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.

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 (Thalheim 2020; Mancas 2020a).

MatBase (Mancas 2018, 2019a, 2020b, 2023) 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, Chen 1976; Thalheim 2000; Mancas 2015), the Relational Data Model (RDM, Codd 1970; Abiteboul, Hull, and Vianu 1995; Mancas 2015), and Datalog (Maier and Warren 1988; Abiteboul, Hull, and Vianu 1995; Mancas 2023).

Currently, MatBase has two versions – one developed in MS Access, for student and small db use and a professional one, developed in MS C# and SQL Server.

This paper presents the pseudocode algorithms used by both MatBase versions to automatically generate code for enforcing dyadic relationship constraints.

2. Related work

This paper is a continuation of Mancas 2020b, which was mainly focused on assisting the process of detecting dyadic relationship constraints. It refines its AEDRC Algorithm (which is a very high level one, mainly dealing with the coherence and minimality of the sets of dyadic relationships constraints) for each type of dyadic relationship constraints.

Other approaches related to the (E)MDM are based on business rules management (BRM) (e.g., von Halle 2001; Morgan 2002; Weiden et al. 2002; Ross 2003; von Halle and Goldberg 2006; Taylor 2019), their corresponding implemented systems (BRMS), and process managers (BPM), like the IBM Operational Decision Manager (Kolban 2015), IBM Business Process Manager (Dyer et al. 2019), Red Hat Decision Manager (Red Hat 2020), Agiloft Custom Workflow/BPM (Agiloft 2020), 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, 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.
3. Prerequisites

Let \( R = (f \rightarrow C, g \rightarrow C) \) be an arbitrary dyadic relationship. For enforcing dyadic relationship type constraints on \( R \), both \( C \) and \( R \) must have Graphic User Interface (GUI) forms associated to their corresponding tables and event-driven methods:

- Classes \( C \) and \( R \) must contain private \( \text{AfterInsert}(x) \) and \( \text{AfterInsert}(f, g) \) methods, respectively (see Figure 3);
- Class \( R \) must contain:
  o Definition of two private numerical variables \( fOldValue \) and \( gOldValue \) (see Figure 1);
  o A private method \( \text{Current}(f, g) \) shown in Figure 1, to be called each time the cursor of the \( R \)’s form enters a new element (line, row, record) of its underlying data;
  o A private method \( \text{BeforeInsert}(f, g) \) shown in Figure 2, to be called each time end-users ask for adding a new element to \( R \);
  o A private \( \text{BeforeUpdate}(f, g) \) method shown in Figure 4, to be called each time a new or existing element of its underlying data whose values for columns \( <f, g> \) were \(<fOldValue, gOldValue>\) and that were then modified to \(<u, v>\) is about to be saved in the db;
  o A private \( \text{AfterUpdate}(f, g) \) method shown in Figure 5, to be called each time an existing element of its underlying data whose values for columns \( <f, g> \) were \(<fOldValue, gOldValue>\) were then modified to \(<u, v>\) and successfully saved to the db;
  o A private \( \text{Delete}(f, g) \) method shown in Figure 6, to be called each time end-users ask for the deletion of an existing element of its underlying data;
  o A private \( \text{AfterDelSuccess}(f, g) \) method shown in Figure 7, to be called each time an existing element of its underlying data whose values for columns \( <f, g> \) were \(<fOldValue, gOldValue>\) were successfully deleted from the db.

All these methods and variables are automatically generated by \textit{MatBase} the first time it needs them.
4. Enforcing connectivity constraints

According to the connectivity definition, enforcing such constraints for \( R \) requires that:

1. Each time a new element \( x \) is added to \( C \), pairs \( <x, y> \) or \( <y, x> \) must be automatically added to \( R \), for any other element \( y \) of \( C \). Moreover, whenever \( R \) is also symmetric, both these pairs should be added.
2. Each time a pair \( <x, y> \) of \( R, y \neq x \), is modified in \( <u, v> \), with either \( u \neq x \) or \( v \neq y \), and there is no \( <y, x> \) in \( R \), then either \( <x, y> \) or \( <y, x> \) must be automatically added to \( R \). Moreover, whenever \( R \) is also symmetric, no such pair should ever be modified.
3. No pair \( <x, y> \) of \( R, y \neq x \), should be deleted, if there is no pair \( <y, x> \) in \( R \). Moreover, whenever \( R \) is also symmetric, no such pair should ever be deleted.

Consequently, MatBase adds the pseudocode algorithms from Figures 8a or 8b to the method AfterInsert of class \( C \) from Figure 3 for case 1, the ones from Figures 9a or 9b to method AfterUpdate from Figure 5 for case 2, and the ones from Figures 10a or 10b to method Delete from Figure 6 for case 3.

---

**Figure 6** Method *Delete* of class \( R \)

**Figure 7** Method *AfterDeleteSuccess* of class \( R \)

---

### Method *Delete*(\( f, g \))

```plaintext
Boole Cancel = False;
if Cancel then
    deny deletion of \(<f,g>\) from \( R \);
```

### Method *AfterDeleteSuccess*(\( f, g \))

---

---

---

### Figure 8a
Code added in method *AfterInsert* of class \( C \) from Figure 3 if \( R \) is not symmetric

```plaintext
// R connected
loop for all \( g \) in \( C, g \neq x \)
    add \(<x, g>\) to \( R \);
end loop;
```

### Figure 8b
Code added in method *AfterInsert* of class \( C \) from Figure 3 if \( R \) is symmetric too

```plaintext
// R connected and symmetric
loop for all \( g \) in \( C, g \neq x \)
    add \(<x, g>\) and \(<g, x>\) to \( R \);
end loop;
```

### Figure 9a
Code added in method *AfterUpdate* from Figure 5 if \( R \) is not symmetric

```plaintext
// R connected
if \( f \neq f\text{OldValue} \) and (\( f \neq f\text{OldValue} \) or \( g \neq g\text{OldValue} \)) and \(<g\text{OldValue}, f\text{OldValue}> \notin R \) then
    add \(<f\text{OldValue}, g\text{OldValue}>\) to \( R \); \( \text{INS} = \text{True} \);
end if;
```

### Figure 9b
Code added in method *AfterUpdate* from Figure 5 if \( R \) is symmetric too

```plaintext
// R connected and symmetric
if \( f \neq f\text{OldValue} \) and (\( f \neq f\text{OldValue} \) or \( g \neq g\text{OldValue} \) then
    Cancel = True; display "Request rejected: \( R \) is both connected and symmetric!";
end if;
```

---

### Figure 10a
Code added in method *Delete* from Figure 6 if \( R \) is not symmetric

```plaintext
// R connected
if \(<g, f> \notin R \) then Cancel = True;
```

### Figure 10b
Code added in method *Delete* from Figure 6 if \( R \) is symmetric too

```plaintext
// R connected and symmetric
Cancel = True;
```
5. Enforcing reflexivity constraints

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

1. Each time a new element $x$ is added to $C$, a pair $<x, x>$ must automatically be added to $R$.
2. No pair $<x, x>$ of $R$ may be modified.
3. No pair $<x, x>$ of $R$ should ever be deleted, unless $x$ is deleted from $C$.

Consequently, MatBase adds the pseudocode algorithm from Figure 11 to the method AfterInsert of class $C$ from Figure 3 for case 1, the one from Figure 12 to method BeforeUpdate from Figure 4 for case 2, and the one from Figure 13 to method Delete from Figure 6 for case 3.

6. Enforcing irreflexivity constraints

According to the irreflexivity definition, enforcing such constraints for $R$ requires that each time a pair $<x, x>$ (be it new or obtained by modifying an existing $<u, v>$) is about to be saved in the db $R$'s image, saving must be canceled.

Consequently, MatBase adds the pseudocode algorithm from Figure 14 to the method BeforeUpdate from Figure 4.

7. Enforcing symmetry constraints

According to the symmetry definition, enforcing such constraints for $R$ not connected (the case $R$ connected is dealt with in section 4) requires that:

1. Each time a pair $<x, y>$, $x \neq y$, is added to $R$, a pair $<y, x>$ must automatically be added to $R$ as well.
2. Each time a pair $<x, x>$ of $R$ is modified in $<u, v>$, with $u \neq v$ and either $u < x$ or $v < x$, then $<v, u>$ must automatically be added to $R$; each time a pair $<x, y>$ of $R$, $y = x$, is modified in $<u, v>$, with $u \neq v$ and either $u < x$ or $v < y$, then $<y, x>$ must automatically be replaced in $R$ by $<v, u>$, whenever $R$ is not connected; and each time a pair $<x, y>$ of $R$, $y = x$, is modified in $<u, v>$ and either $u < x$, or $v < y$, then $<y, x>$ must automatically be deleted from $R$, whenever $R$ is not connected.
3. Each time a pair $<x, y>$ of $R$, $y = x$, is deleted, then $<y, x>$ must automatically be deleted from $R$ as well, whenever $R$ is not connected.

Consequently, whenever $R$ is not connected, MatBase adds the pseudocode algorithm from Figure 15 to the method AfterInsert of class $R$ from Figure 3 for case 1, the one from Figure 16 to method AfterUpdate from Figure 5 for case 2, and the one from Figure 17 to method AfterDelSuccess from Figure 7 for case 3.
8. Enforcing asymmetry constraints

According to the asymmetry definition, enforcing such constraints for $R$ requires that:

1. Each time a pair $<x, y>, x \neq y$, is about to be added to $R$, this must be rejected whenever a pair $<y, x>$ exists in $R$.

2. Each time a pair $<x, y>$ of $R$ is modified in $<u, v>$, with $u \neq v$ and either $u \neq x$ or $v \neq y$, this must be rejected whenever a pair $<v, u>$ exists in $R$.

Consequently, MatBase adds the pseudocode algorithm from Figure 18 to the method BeforeInsert of class $R$ from Figure 2 for case 1 and the one from Figure 19 to the method BeforeUpdate from Figure 4 for case 2.

9. Enforcing transitivity constraints

According to the transitivity definition, enforcing such constraints for $R$ requires that:
1. Each time a pair \( <x, y> \), \( x \neq y \), is added to \( R \) and \( R \) contains a pair \( <y, z> \), \( z \neq y \), a pair \( <x, z> \) must automatically be added to \( R \) as well, if it does not exist already.

2. Each time a pair \( <x, z> \) of \( R \) is modified in \( <u, v> \), with either \( u \neq x \) or \( v \neq z \), and there is at least a \( y \) in \( C \) such that both \( <x, y> \) and \( <y, z> \) belong to \( R \), then modification of \( <x, z> \) must be rejected.

3. Each time a pair \( <x, z> \) of \( R \) is about to be deleted and there is at least a \( y \) in \( C \) such that both \( <x, y> \) and \( <y, z> \) belong to \( R \), then deletion of \( <x, z> \) must be rejected.

Consequently, \( \text{MatBase} \) adds the pseudocode algorithm from Figure 20 to the method \( \text{AfterInsert} \) of class \( R \) from Figure 3 for case 1, the one from Figure 21 to method \( \text{BeforeUpdate} \) from Figure 4 for case 2, and the one from Figure 22 to method \( \text{Delete} \) from Figure 6 for case 3, but only when \( R \) is not connected and symmetric as well (case in which, according to the algorithms for the coherence and minimality of the constraint sets (Mancas 2018, 2020b, 2023), transitivity is redundant, being implied by connectivity and symmetry).

10. Enforcing intransitivity constraints

According to the intransitivity definition, enforcing such constraints for \( R \) requires that:

1. Each time a pair \( <x, z> \) is about to be added to \( R \) and there are at least two pairs \( <x, y> \) and \( <y, z> \) stored by \( R \), then adding \( <x, z> \) to \( R \) must be rejected.

2. Each time a pair \( <u, v> \) of \( R \) is modified in \( <x, z> \), with either \( u \neq x \) or \( v \neq z \), and there is at least a \( y \) in \( C \) such that both \( <x, y> \) and \( <y, z> \) belong to \( R \), with \( y \neq x \) and \( y \neq z \), then modification of \( <u, v> \) must be rejected.

Consequently, \( \text{MatBase} \) adds the pseudocode algorithm from Figure 23 to the method \( \text{BeforeInsert} \) of class \( R \) from Figure 2 for case 1 and the one from Figure 24 to the method \( \text{BeforeUpdate} \) from Figure 4 for case 2.
11. Enforcing Euclideanity constraints

According to the Euclideanity definition, enforcing such constraints for $R$ requires that:

1. Each time a pair $<x, y>$ is added to $R$ and $R$ contains a pair $<x, z>$, a pair $<y, z>$ must automatically be added to $R$ as well, if it does not exist already.
2. Each time a pair $<y, z>$ of $R$ is modified in $<u, v>$, with either $u \neq y$ or $v \neq z$, and there is at least a $x$ in $C$ such that both $<x, y>$ and $<x, z>$ belong to $R$, then modification of $<y, z>$ must be rejected.
3. Each time a pair $<y, z>$ of $R$ is about to be deleted and there is at least a $x$ in $C$ such that both $<x, y>$ and $<x, z>$ belong to $R$, then deletion of $<y, z>$ must be rejected.

Consequently, MatBase adds the pseudocode algorithm from Figure 25 to the method AfterInsert of class $R$ from Figure 3 for case 1, the one from Figure 26 to method BeforeUpdate from Figure 4 for case 2, and the one from Figure 27 to method Delete from Figure 6 for case 3, but only when $R$ is not connected and symmetric as well (case in which, according to the algorithms for the coherence and minimality of the constraint sets (Mancas 2018, 2020b, 2023), Euclideanity is redundant, being implied by connectivity and symmetry).

12. Enforcing inEuclideanity constraints

According to the inEuclideanity definition, enforcing such constraints for $R$ requires that:

1. Each time a pair $<y, z>$ is about to be added to $R$ and there are at least two pairs $<x, y>$ and $<x, z>$ stored by $R$, then adding $<y, z>$ to $R$ must be rejected.
2. Each time a pair $<u, v>$ of $R$ is modified in $<y, z>$, with either $u \neq y$ or $v \neq z$, and there is at least a $x$ in $C$ such that both $<x, u>$ and $<x, v>$ belong to $R$, with $y \neq x$ and $y \neq z$, then modification of $<u, v>$ must be rejected.

Consequently, MatBase adds the pseudocode algorithm from Figure 28 to the method BeforeInsert of class $R$ from Figure 2 for case 1 and the one from Figure 29 to the method BeforeUpdate from Figure 4 for case 2.

![Figure 25 Code added in method AfterInsert from Figure 3](image)

![Figure 26 Code added in method BeforeUpdate from Figure 4](image)

![Figure 27 Code added in method Delete from Figure](image)

![Figure 28 Code added in method BeforeInsert from Figure 2](image)

![Figure 29 Code added in method BeforeUpdate from Figure 4](image)
13. Enforcing equivalence constraints

According to a definition of relation equivalence, enforcing it for $R$ requires that $R$ be both reflexive and Euclidean. Consequently, equivalence is be enforced by merging the algorithms from sections 5 and 11.

14. Enforcing acyclicity constraints

According to the acyclicity definition, enforcing such constraints for $R$ requires that:

1. Each time a pair $<x, y>$ is about to be added to $R$ and there is a path of pairs $<y, x_1>, ..., <x_n, x>, n>0$, exists in $R$, then adding $<x, y>$ to $R$ must be rejected.
2. Each time a pair $<u, v>$ of $R$ is modified in $<x, y>$, with either $u \neq x$ or $v \neq y$, this must be rejected whenever a path of pairs $<x_1, x>, ..., <x_n, x>, n>0$, exists in $R$.

Consequently, MatBase adds the pseudocode algorithm from Figure 30 to the methods BeforeInsert of class $R$ from Figure 2 for case 1 and BeforeUpdate from Figure 4 for case 2. For detecting the paths that would close cycles in the graphs of any dyadic relationship $R$, MatBase uses the corresponding Dijkstra algorithm (Dijkstra 1959; Mancas 2023). The Boolean function $DIJKSTRA$ that implements it takes as parameters the names of the table storing the dyadic relationship $R$ and of its columns ($f$, $g$, and $x$, for its primary key), as well as the current values for its canonical Cartesian projections $f$ and $g$, and returns True if such a path exists or False otherwise.

```
// R acyclic
if not Cancel then Cancel = DIJKSTRA("R","x","g","f","f");
if Cancel then display "Request rejected: R acyclic";
```

Figure 30 Code added in methods BeforeInsert from Figure 2 and BeforeUpdate from Figure 4

15. The MatBase algorithm for enforcing above constraints

Figures 31 to 35 show the MatBase algorithm for enforcing dyadic relationship constraints.

16. Concluding remarks and implications for future research

It is straightforward to check that applying the Algorithm $A9DR$ from Figure 30 to $C = TEAMS$ and $R = MATCHES$ from the Introduction section, MatBase automatically generates for their corresponding classes the pseudocode shown in Figures 36 to 40.

Let us suppose that the TEAMS and MATCHES tables are empty and that end-users start to enter data for the 2021-2022 UK soccer Premier League in this db. Suppose that they start by adding Manchester City to TEAMS (which automatically gets 1 in its $x$ primary key column); when saving it, method AfterInsert from Figure 35 is automatically invoked, but does nothing, as there is no other team in TEAMS. When end-users save then, say, Liverpool (which gets $x = 2$), method AfterInsert from Figure 35 automatically inserts $<2, 1>$ and line $<1, 2>$ in MATCHES (which corresponds to the matches $<Liverpool, Manchester City>$ and $<Manchester City, Liverpool>$, respectively). When end-users save then, say, Chelsea (which gets $x = 3$), method AfterInsert from Figure 35 automatically inserts $<3, 1>$, $<1, 3>$, $<2, 3>$ and $<3, 2>$ in MATCHES (which corresponds to the matches $<Chelsea, Manchester City>$, $<Manchester City, Chelsea>$, $<Manchester City, Liverpool>$, and $<Manchester City, Liverpool>$, respectively). Obviously, when all the 20 teams are saved in TEAMS, all corresponding 38 matches between them are automatically saved in MATCHES.

As MATCHES is both connected and symmetric, method Delete from Figure 37 prevents deletions from this table. Method BeforeUpdate from Figure 39 prevents inserting in MATCHES lines of type $<Host, Host>$, as well as modifying any $<Host, Visitor>$ one.

To conclude with, the above automatically generated code by MatBase is both guaranteeing data plausibility in this db and automatically generating the instance of the MATCHES table while end-users are entering data into the TEAMS one.
MatBase algorithm A9DR for enforcing dyadic relationship constraints

Input: A db software application SA over a dyadic relation \( R = (f \rightarrow C, g \rightarrow C) \) and a constraint \( c \) of subtype \( s \) on \( R \)
Output: SA augmented such as to enforce \( c \) as well

Strategy:
switch (s)
    case s: connectivity
        if \( R \) is symmetric then add to method AfterInsert of class \( C \) from Figure 3 the code from Figure 8b, to the
        AfterUpdate one of class \( R \) from Figure 5 the code from Figure 9b, and to the Delete one of class \( R \) from Figure
        6 the code from Figure 10b;
        else add to method AfterInsert of class \( C \) from Figure 3 the code from Figure 8a, to the AfterUpdate one of class \( R \)
        from Figure 5 the code from Figure 9a, and to the Delete one of class \( R \) from Figure 6 the code from Figure 10a;
        if \( R \) is transitive then enforceTransitivity;
        if \( R \) is Euclidean then enforceEuclideanity;
        end if;
        if \( R \) is reflexive then enforceReflexivity;
        break;
    case s: reflexivity
        enforceReflexivity;
        break;
    case s: irreflexivity
        add to method BeforeUpdate of class \( R \) from Figure 4 the code from Figure 14;
        break;
    case s: symmetry
        if not connected then
            enforceSymmetry;
            if \( R \) is transitive then enforceTransitivity;
            if \( R \) is Euclidean then enforceEuclideanity;
        end if;
        break;
    case s: asymmetry
        add to method BeforeInsert of class \( R \) from Figure 2 the code from Figure 18 and to the BeforeUpdate one of class
        \( R \) from Figure 4 the code from Figure 19;
        break;
    case s: transitivity
        if \( R \) is not both connected and symmetric then
            enforceTransitivity;
            if \( R \) is symmetric then enforceSymmetry;
            if \( R \) is Euclidean then enforceEuclideanity;
        end if;
        break;
    case s: intransitivity
        add to method BeforeInsert of class \( R \) from Figure 2 the code from Figure 23 and to the BeforeUpdate one of class
        \( R \) from Figure 4 the code from Figure 24;
        break;
    case s: Euclideanity
        if \( R \) is not both connected and symmetric then
            enforceEuclideanity;
            if \( R \) is symmetric then enforceSymmetry;
            if \( R \) is transitive then enforceTransitivity;
        end if;
        break;

Figure 31 MatBase algorithm A9DR for enforcing dyadic relationship constraints
Generally, the Algorithm $A9DR$ from Figure 30 automatically generates code that is guaranteeing data plausibility for any dyadic relationship for which all its properties are declared to $\text{MatBase}$ as corresponding constraints, while also automatically generating its core data values, thus saving most of the developing, testing, and data entering effort.

Moreover, please note that the (E)MDM also includes constraints on self-maps and binary homogeneous function products (Mancas 2018, 2019b, 2023), which are particular cases of dyadic relationships.

For example, the self-map $\text{Mother} : \text{PERSONS} \rightarrow \text{PERSONS}$ is irreflexive (i.e., nobody may be his/her mother), asymmetric (i.e., no mother may be the child of his/her child), intransitive (i.e., anti-idempotent, because if $y$ is the mother of $x$, then $y$ may not be his/her own mother), and acyclic (i.e., nobody may be his/her either ancestor or descendant, on no generation level).
For example, the binary homogeneous function product $Mother \cdot Father : PERSONS \rightarrow PERSONS \times PERSONS$ is irreflexive (i.e., nobody may be both your mother and father), asymmetric (i.e., if $y$ is the mother of $x$ and $z$ his/her father, then there may not be any person $p$ having $y$ as mother and $z$ as father), in Euclidean (i.e., if $x$ is the mother of both $u$ and $v$, $y$ is the father of $u$ and $z$ the one of $v$, then there may not be a person $p$ having $u$ as mother and $v$ as father, as $u$ and $v$ are siblings), and acyclic (i.e., no man may give birth and no woman may be a father of a child).

Consequently, future research will be devoted to describing what code is $MatBase$ automatically generated for enforcing the constraints associated with both self-maps and homogeneous binary function products.

17. Conclusion

Not enforcing any existing business rule from the sub-universe managed by a db software application allows saving implausible data in its db. This paper presents the algorithms needed to enforce the 11 possible dyadic relationship constraint types from the (E)MDM, which are implemented in $MatBase$, an intelligent DBMS prototype. Moreover, as $MatBase$ automatically generates the corresponding code, it is a tool of the 5th generation programming languages – modelling as programming: db and software architects only need to assert the properties of the dyadic relationships.
(and not only, but of all other (E)MDM constraint types), while MatBase saves the corresponding developing, testing, and debugging time.

**Compliance with ethical standard**

**Acknowledgement**

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

**Disclosure of conflict of interest**

There is no conflict of interest.

**References**


