MATBASE AUTOFUNCTION NON-RELATIONAL CONSTRAINTS ENFORCEMENT ALGORITHMS

Christian Mancas

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

ABSTRACT

MatBase is an intelligent prototype data and knowledge base management system based on the Relational (RDM), Entity-Relationship, and (Elementary) Mathematical ((E)MDM) Data Models, built upon Relational Database Management Systems (RDBMS). ((E)MDM) has 61 constraint types, out of which 21 apply to autofunctions as well. All five relational (RDM) constraint types are passed by MatBase for enforcement to the corresponding RDBMS host. All non-relational ones are enforced by MatBase through automatically generated code. This paper presents and discusses both the strategy and the implementation of MatBase autofunction non-relational constraints enforcement algorithms. These algorithms are taught to our M.Sc. students within the Advanced Databases lectures and labs, both at the Ovidius University and at the Department of Engineering in Foreign Languages, Computer Science Taught in English Stream of the Bucharest Polytechnic University, as well as successfully used by two Romanian software companies.

KEYWORDS

intelligent systems, data modeling, database constraints theory, relational constraints, non-relational constraints, integrity checking, data structures and algorithms for data management, triggers and rules, business rules, (Elementary) Mathematical Data Model, MatBase, automatic code generation

1. INTRODUCTION

MatBase is an intelligent prototype data and knowledge base management system based on the Relational Data Model [1, 5, 13] (RDM), Entity-Relationship Data Model (E-RDM) [4, 13, 25], and (Elementary) Mathematical Data Model ((E)MDM) [12, 13, 14, 15, 16], built upon Relational Database Management Systems (RDBMS), and currently having two versions: one in VBA on MS *Access* and one in C# on MS *SQL Server* [17].

Any universe of discourse that is interesting for data modeling is made of object sets, their properties, and the business rules that are governing it. Mathematically, object sets are abstracted as sets (be them atomic or relations, i.e. subsets of Cartesian products), properties as functions, and business rules as first order predicate calculus closed formulas. In relational databases (dbs), sets are implemented as tables, functions as their columns, and constraints either as relational ones, enforceable in pure SQL and/or in the Graphic User Interfaces (GUI) of RDBMSes, or as non-relational, enforceable with either extended SQL (generally with triggers) or/and a high-level programming language embedding SQL (generally with trigger-type event-driven procedures).

RDM introduced five types of (relational) constraints: *domain* (range; codomain – mathematically), *not null* (totally defined – mathematically), *key* (uniqueness; minimally one-to-oneness – mathematically), *typed inclusion* (referential integrity, foreign key; function images inclusion – mathematically), and *tuple* (check; closed first order predicate calculus formulas having only one variable universally quantified and whose functions have same domain – mathematically).

DOI: 10.5121/ijcsit.2019.11505

RDBMSes also generally provide a sixth one, which is also provided by *MatBase*, namely column (function – mathematically) *default value*. Both existing *MatBase* versions are enforcing these by simply passing them to their RDBMS hosts [18].

In RDBMSes, autofunctions (i.e. functions of type $f : A \rightarrow B$, with $B \subseteq A$ or $A \subseteq B$) are implemented as foreign keys referencing their table (e.g. in table *PERSONS*, having primary key *x*, both *Mother* and *Father* are referencing *x*). Consequently, only not-null, key, referential integrity, and tuple constraints are applicable to autofunctions, from the RDM point of view (plus the default value, from the RDBMS' one).

From the (E)MDM's one (which also considers compound functions, not only singleton ones), for any function there are also non-primeness (i.e. the function may not be part of any key), ontoness, and bijectivity. As autofunctions are particular cases of dyadic relations, both (null-)reflexivity, (null-)irreflexivity, (null-)symmetry, asymmetry, (null-)idempotence, anti-idempotence, canonical surjectivity (ontoness), and acyclicity are also applicable (while (in)Euclidianity and connectivity are not, as they do not make sense for functions [14]). Moreover, general object constraints [14, 16] are also possible for autofunctions too. This paper focuses on the algorithms used by *MatBase* to enforce these latter 16 (non-relational) constraint types.

For example, such a non-relational constraint exists even in an extremely simple db consisting only of the tables *STATES*(*x*, *Country*, *State*, *StateCapital*) and *CITIES*(*x*, *City*, *State*): "any state has as capital a city of its own" (or, dually, "no state may have as its capital a city of another state"). Considering functions *State* : *CITIES* → *STATES* and *StateCapital* : *STATES* → *CITIES*, this constraint can be formalized as *State* ° *StateCapital* = $\mathbf{1}_{STATES}$ (where ° denotes function composition and $\mathbf{1}_{STATES}$ is the unity mapping of *STATES*, $\mathbf{1}_{STATES}(x) = x$, $\forall x \in STATES$) or, equivalently, as *State* ° *StateCapital reflexive*. This constraint is associated to the E-RD cycle from Figure 1 (where *StateCapital* has a double arrow because it is one-to-one).

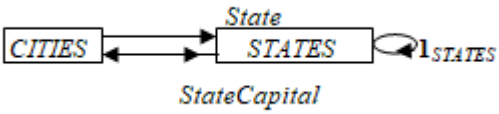

Figure 1. An example of an E-RD cycle having an associated non-relational constraint.

Failing to enforce this constraint could result, for example, in letting users store in such a db instance the fact that London is the capital of Romania's state Teleorman and/or that the Romanian city Caracal is the capital of the U.S.' state California!

From *MatBase* users' point of view, all above mentioned first 15 non-relational constraint types may be added / removed from a db scheme by simply clicking the corresponding check-boxes of its *FUNCTIONS* form (be it standalone or as a subform of the *SETS_DETAILED* one), available in its *Math Scheme* submenu.

Removing such a constraint is straightforward: if it is implied by the rest of the constraints (e.g. if *f* is asymmetric, you cannot remove its irreflexivity), *MatBase* does not allow it, while otherwise it does, which means that the corresponding code enforcing that constraint is deleted from the class(es) associated to the form(s) that manage(s) corresponding data (i.e. built upon the table corresponding to the autofunction domain).

Detecting that a constraint is implied by the rest of the constraint set to which it belongs, as well as whether adding it to a constraint set would make that set incoherent is done by using the *MatBase* knowledge base and algorithm presented in [14].

Whenever adding a constraint preserves corresponding constraint set's coherence, *MatBase* automatically injects the needed code for enforcing it into the class(es) associated to the form(s) that manage(s) the corresponding data (that are automatically generated by *MatBase* when the corresponding domain set is added to the db scheme). Composed functions (e.g. *State* ° *StateCapital* above) may be either explicitly specified by users or automatically generated by *MatBase* whenever it is detecting cycles in a db E-RD [16, 19].

As another example, even for only one table, e.g. *PERSONS*(*x*, *FirstName*, *LastName*, *Mother*, *Father*, *Sex*, *BirthDate*, *PassedAwayDate*), where *Mother* and *Father* are acyclic autofunctions, non-relational general object-type constraints may exist as well: mothers should have sex 'F', fathers 'M', no child may be born either after his/her mother's death, or more than 9 months after his/her father's death, etc. Constraints of this general type are expressible in *MatBase* in its *OBJECT_CONSTRAINTS* form, where users are expected to type the corresponding formula in a user-friendly first-order predicate calculus structured language. For the time being, neither coherence, nor minimality is automatically enforced by *MatBase* for this object constraint type, but left to its users.

MatBase, just like most of the db applications, is presenting autofunctions to users as combo-boxes in the form(s) built over the table corresponding to their domain. Generally [15], whenever possible, non-relational constraints should be enforced preventively, by eliminating from these combo-boxes all implausible data. When this is not possible [15], they should be enforced curatively through extended SQL triggers or/and SQL embedding trigger-type methods.

For example, the reflexivity constraint associated to Figure 1 above is preventively enforced partially with only two VBA / C# statements of the *Current* event-driven method associated to cursor entering on a line: the first one dynamically changes the WHERE clause of the *StateCapital* combo-box of the form(s) based on the *STATE* table, such that only cities belonging to the current state are selected, and the second one re-queries the combo-box to apply its changed definition. However, this constraint might be also violated whenever users were trying to move capital cities to another state: therefore, *MatBase* also automatically generates code in the *BeforeUpdate* (for its MS Access version) and *Validating* (for its MS .NET version) event-driven methods attached to the *State* combo-box of class *CITIES* form(s) to reject such requests.

1.1 Related Work

Generally, there are very few results on non-relational constraints, except for (E)MDM related ones ([12, 14, 15, 16,19]). The most closely related approaches are based on business rules management (BRM) [8, 9, 20, 24, 27] and their corresponding implemented systems (BRMS) and process managers (BPM), like *IBM Operational Decision Manager*[10], *IBM Business Process Manager* [6], *Red Hat Decision Manager* [21], *Agiloft Custom Workflow/BPM* [2], etc., generally based on XML (but also on the Z notation, Business Process Execution Language, Business Process Modeling Notation, Decision Model and Notation, or the Semantics of Business Vocabulary and Business Rules), which is the only other field of endeavor trying to systematically deal with business rules, even if informally, not at the db design level, but at the software application one, and without providing automatic code generation. 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 [7, 11, 22, 23, 26], which aims only at solving polynomial-complexity combinatoric problems (e.g. planning, scheduling, etc.) [3].

1.2 Paper Outline

Section 2 introduces *MatBase* architecture. Section 3 describes the relevant portions of its Graphical User Interface (GUI). Section 4, the paper's core, presents the algorithms for enforcing autofunction non-relational constraints. Section 5 illustrates them with examples. Section 6 discusses their complexity, optimality, implementation, and utility. The paper ends with conclusion, further work, and references.

2. *MATBASE* ARCHITECTURE

MatBase has a standard 4-tier architecture, whose tiers are (in the top-down order) the following: GUI, business logic (BL), ActiveX Data Objects (ADO), and db (DB) [17].

The MS *Access* version DB is composed of two shared dbs, namely MatBaseDB.accdb and GeographyDB.accdb (stored on a file server folder, identified through network mapping as virtual logic drive V:), and each workstation storing eight other dbs, namely MatBaseTmp.accdb, GeographyTmp.accdb, StocksDB.accdb, StocksTmp.accdb, BookstoreDB.accdb, BookstoreTmp.accdb, UserDB.accdb, and UserTmp.accdb (stored in a folder declared, through *subst*, as being virtual logic drive U:).

MatBaseDB contains the *Matbase*'s metacatalog (storing metadata on the managed dbs, tables, constraints, etc.) and knowledge base (storing data on coherence and minimality of constraint sets, Datalog inference rules, object constraint type ones, etc.). As *MatBase* also provides four db application examples (*Geography*, shareable, *Stocks*, *Bookstore*, and *User*) there are also db files storing their data. All needed temporary tables are stored in corresponding Tmp dbs.
In its MS SQL Server versions, all these 5 fundamental dbs are stored on a server instance, whereas all temporary ones in the corresponding system tempDB db.

ADO is the (de facto industry standard) middleware between BL and DB, completely transparent to programmers in the .net and SQL Server *MatBase* version, carrying SQL statements as strings from BL to DB and returning error codes and selected data. In the Access version, ADO is also explicitly used, as, for example, there is no CHECK constraint in its SQL.

For its both versions, BL is made of object-oriented classes (most of them also event-driven, associated to forms and reports) and libraries grouping methods commonly used by at least two classes. Every db application has its own library (e.g. *Geography*, *Stocks*, *Bookstore*, etc.). *MatBase*'s core has several specialized ones: *Constraints*, *Datalog*, *ERD*, *General*, *Mappings*, *Sets*, and *Tools*, with *General* including methods, variables, and constants commonly used by at least two other libraries.

For example, the *Constraints* one includes parameterized Boolean functions for enforcing object constraints (*enforceObjConstraint*), autofunction reflexivity (*autoFunctReflex*), irreflexivity (*autoFunctIrreflex*), etc. Whenever the current data passed as parameters satisfies the corresponding constraint, these functions return *False*, otherwise they return *True* (following the event-driven methods' *Cancel* parameter conventions).

Whenever enforcing of a non-relational constraint is possible (i.e. it would not violate coherence or minimality of the corresponding constraint set and the current db instance satisfies it), *MatBase* injects in the corresponding *BeforeUpdate* / *Validating* event-driven method associated to the corresponding autofunction (in the object-oriented class(es) of the form(s) associated with the table corresponding to the autofunction domain set) an assignment of type *Cancel = call to the corresponding Boolean function from Constraints*; therefore, if the returned value is *False*, then the corresponding data update is accepted and saved in the db; otherwise, it is rejected, with a corre-

sponding context-dependent error message inviting users to modify it or cancel the current request.

Constraints also includes Boolean functions *isCoherent* (checking that adding / removing a constraint satisfies or violates coherence), *isValid* (checking whether the current db instance satisfies a given constraint), *injectConstraint* (inserting code lines into an event-driven class), *deleteModuleLines* (deleting all lines in a class exactly containing a string parameter value), as well as many other useful functions for automatically enforcing constraints, like *findModuleLines* (returning the first line number in a class containing a string parameter value), etc.

GUI includes a tree-like menu, forms, and reports, all of them carefully designed, as user-friendly as possible. Each fundamental db table has at least one standard associated form, automatically generated, which manages corresponding data management (i.e. inserts / updates / deletes).

3. MATBASE GUI FOR AUTOFUNCTION AND OBJECT CONSTRAINTS

In the *MatBase MetaCatalog* / *Scheme Updates* / *EMDM* Scheme submenu, the form *FUNCTIONS* may be opened either standalone (with the *Mapping Set* button of the *Mappings* submenu), for managing all functions known to the system, or as a subform of the *SETS_DETAILS* form (opened with the button *Update SETS set by set* button of the *Sets* submenu), for managing functions grouped by their domain set; the form *OBJECT_CONSTRAINTS* may be opened with the homonym button from the submenu *Constraints* of the same menu path.

The form *FUNCTIONS* displays one row per function (be it fundamental or computed). For composed ones, another instance of it may be opened as a subform, displaying a line for each member function, in their composition order. All metadata on functions is displayed and may be updated in its column controls, from its name, domain, codomain, description, and up to its constraints (except for the object-type ones). For constraint types, check-boxes are used, except for the default value one, which needs a text box. For example, for *State* ° *StateCapital* from Figure 1 only the *reflexive* check-box should be checked; for *Mother* and *Father*, only the *acyclic* one should be checked, which triggers *MatBase* to automatically check the implied constraints *asymmetric*, *irreflexive*, and *anti-idempotent* as well.

The form *OBJECT_CONSTRAINTS* displays one row per object constraint. For example, there should be a line reading "for any x in PERSONS Sex(Mother(x)) = 'F'" (which formalizes "mothers should have sex 'F'"), another one reading "for any x in PERSONS Sex(Father(x)) = 'M'" (which formalizes "fathers should have sex 'M'"), one reading "for any x in PERSONS BirthDate(x) <= PassedAwayDate(Mother(x)) and BirthDate(x) <= PassedAwayDate(Father(x)) + 9 * 30.5" (which formalizes "no child may be born either after his/her mother's death, or more than 9 months after his/her father's death"), etc.

4. *MATBASE* ALGORITHMS FOR ENFORCING AUTOFUNCTION AND OBJECT-TYPE NON-RELATIONAL CONSTRAINTS

Figures 2 and 3 show the pseudocode algorithms for enforcing object and autofunction non-relational constraints, respectively, which summarizes all the above and that are implemented in both current *MatBase* versions:

ALGORITHM *A1*. *MatBase* Algorithm for Enforcing Object Constraints

---

Input: - a db scheme *S*, its associated *OBJECT_CONSTRAINTS* form instance,
the user request (delete / insert / update), and the corresponding object

constraints from its current row *c* and *c'*, the former one updated to *c*;
- *Cancel = False*;

Output: **if** *Cancel* **then** *S* **else if** delete **then** $S = S - \{c\}$ **else if** insert **then** $S = S \cup \{c\}$
else $S = S - \{c'\} \cup \{c\}$;

Strategy:
select user request
case delete:
if user does not confirm his/her delete request **then** *Cancel = True*;
else
loop for all object sets *s* mentioned in the current object constraint *c*
loop for all event-driven methods of the classes associated to *s*
delete line assigning to *Cancel* the value returned by *objConstraint* for *c*;
end loop;
end loop;
$S = S - \{c\}$;
if there is no other object constraint of the type *t* of *c* **then**
if user agrees with it **then** delete *t* from the knowledge base;
end if;
end if;
case insert or update:
Cancel = isValid(*c*);
if *Cancel* **then** display "Constraint cannot be enforced: current db instance violates it!";
else
if *c* type is not known to the knowledge base **then** save its type; // which also generates
// and saves in *Constraints* a corresponding method for enforcing it;
end if;
end if;
case insert:
if not *Cancel* **then**
loop for all object sets *s* mentioned in the current object constraint *c*
loop for all event-driven methods of the classes associated to *s* that are stored in the
knowledge base for the type of *c* (and generate all those that might be missing)
inject line assigning to *Cancel* the value returned by *objConstraint* for *c*;
end loop;
end loop;
$S = S \cup \{c\}$;
end if;
case update:
if not *Cancel* **then**
loop for all object sets *s* mentioned in the current former object constraint *c'*

loop for all event-driven methods of the classes associated to *s*
delete line assigning to *Cancel* the value returned by *objConstraint* for *c'*;
end loop;
end loop;
loop for all object sets *s* mentioned in the current object constraint *c*
loop for all event-driven methods of the classes associated to *s* that are stored in the knowledge base for the type of *c* (and generate all those that might be missing)
inject line assigning to *Cancel* the value returned by *objConstraint* for *c*;
end loop;
end loop;
$S = S - \{c'\} \cup \{c\}$;
end if;
end select;
End ALGORITHM *A*1;

Figure 2. Algorithm *A*1 (*MatBase* Algorithm for Enforcing Object Constraints)

ALGORITHM *A*2. *MatBase* Algorithm for Enforcing Non-Relational Autofunction Constraints

Input: - a db scheme *S*, its associated *FUNCTIONS* form instance, the user request (check / un check), the corresponding function $f: D \to D$, 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** $S = S \cup \{c\} \cup I$;
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 object sets *s* that are domains of the functions *g* composing *f*
loop for all event-driven methods of the classes associated to *g*
delete line assigning to *Cancel* the value returned for *c* by the corresponding constraint type enforcement Boolean function;
end loop;

```
            end loop;
            S = S — {c};
            loop for all constraints c' in S that were implied only by c
              if user wishes to keep c' in S 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 object sets s that are domains of the functions g composing f
               loop for all event-driven methods of the classes associated to g (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;
             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 A2;
```

Figure 3. Algorithm *A*2 (*MatBase* Algorithm for Enforcing Non-Relational Autofunction Constraints)

5. EXAMPLES

Consider the ER-D from the *Geography* db presented in Figure 1. Suppose that the db instance data satisfies constraint *State* ° *StateCapital reflexive* and that adding this constraint does not violate either coherence or minimality of the db constraint set [14].

Then, when users check the *Reflexive* check box on the *State* ° *StateCapital* row from the *FUNCTIONS* form, *MatBase* automatically adds code to both *CITIES* and *STATES* classes of their standard management forms (generally homonyms to their underlying table names), as well as to any other classes associated to forms built upon tables *CITIES* and *STATES* (that have, just like all other fundamental tables, primary keys named *x*) and allowing updates to columns *State* and/or *StateCapital*, the following code:

- In class *STATES*, preventively, to the *Current* event-driven method (automatically launched by the system whenever the cursor arrives on a line of the underlying table *STATES*), a line reading Cancel = preventReflex("State =", Me!x) (in the MS Access version; in the .Net one, Me!x is replaced by this.x) is added (and if this event-driven method does not exist, *MatBase* first creates it).
  The public *preventReflex* function of the *Constraint* library has only two lines: the first one modifies the row (data) source definition of the current combo-box control by adding to its WHERE clause a filter "State = " & Me!x and the second one requeries the combo-box.
  In this case, if the initial row source definition is "SELECT x, City FROM CITIES ORDER BY City", the modified one becomes "SELECT x, City FROM CITIES WHERE State = " & Me!x & " ORDER BY City". Without this filter, on any row, the combo-box displays all cities from table CITIES. With this filter, whenever the cursor arrives on a row for which, e.g. $x = 1$, the row source is changed to "SELECT x, City FROM CITIES WHERE State = 1 ORDER BY City", so, after re-querying it, the combo-box will display only the cities that belongs to the state having its $x = 1$.

- As this constraint may also be violated by moving a state capital to another state, such attempts must be rejected for such cities. This is why, in class *CITIES*, to the *BeforeUpdate* (*Validating* in .Net) event-driven method associated to the column control *State* (automatically launched by the system whenever its value has changed and users want to save its new value), a line reading Cancel = enforceReflex("STATES", "StateCapital", Me!x) is added (and if this event-driven method does not exist, *MatBase* first creates it).
  The public *enforceReflex* function from the *Constraint* library has the following strategy:
  enforceReflex = *False*;
  if not new record **then**
  if current city is a state capital **then**
  enforceReflex = *True*;
  undo user update;
  display "Current city is the state capital: request rejected!"
  end if;
  end if;

Dually, when users uncheck the *Reflexive* check-box on the *State* ° *StateCapital* row from the *FUNCTIONS* form, if users confirm their request *MatBase* automatically deletes the code it injected (i.e. both Cancel = preventReflex("State =", Me!x) from *STATES* and Cancel = enforceReflex("STATES", "StateCapital", Me!x) from *CITIES*).

As another example, suppose that a user adds in the *OBJECT_CONSTRAINTS* forms a row whose *Constraint* text box reads "for any x in PERSONS BirthDate(x) <= PassedAwayDate(Mother(x)) and BirthDate(x) <= PassedAwayDate(Father(x)) + 9 * 30.5" and that the current db instance sat-

isfies it. Then, *MatBase* automatically adds in the classes attached to the forms based on the *PERSONS* table that allow updates to the *BirthDate*, *PassedAwayDate*, *Mother*, and/or *Father* the following code (supposing that this constraint gets the *x* identification value 1):

- In the event-driven method *BeforeUpdate* / *Validating* associated to the *BirthDate* control a line reading Cancel = enforceObjConstraint(1, "PERSONS", "BirthDate");
- In the event-driven method *BeforeUpdate* / *Validating* associated to the *PassedAwayDate* control a line reading Cancel = enforceObjConstraint(1, "PERSONS", "PassedAwayDate");
- In the event-driven method *BeforeUpdate* / *Validating* associated to the *Mother* control a line reading Cancel = enforceObjConstraint(1, "PERSONS", "Mother");
- In the event-driven method *BeforeUpdate* / *Validating* associated to the *Father* control a line reading Cancel = enforceObjConstraint(1, "PERSONS", "Father");
- If any of these event-driven methods does not exist, *MatBase* first creates it.

The public Boolean *enforceObjConstraint* function from the *Constraint* library (one of the most complex ones) enforces such constraints according to their corresponding formula type and given parameters (object constraint id, set / table name, and function / column name).

Dually, when users delete this object constraint from the *OBJECT_CONSTRAINTS* form, *MatBase* deletes all four lines injected as above.

6. RESULTS AND DISCUSSION

6.1 Algorithms Complexity, Optimality, and Implementation

It is very easy to check that these algorithms are very fast, as they do not ever infinitely loop and their time complexities are linear: $O(2 * k * (|F|)) = O(|F|)$ for $A1$ and $O(k * (|F| + |I|)) = O(|F| + |I|)$ for $A2$. For both of them, k is the average number of forms built over fundamental tables (which is generally between 1 and 2).

For $A1$, $|F|$ is the average number of functions used in an object constraint (multiplied by 2 in the worst case, i.e. update, for which *MatBase* needs to both inject code for c and delete code for c'); generally, this never approaches 100 (conjecture we are advancing after 40+ years of experience in data modeling for lot of industries and services).

For $A2$, $|F|$ is the average number of functions that are members of a composed function (generally at most 8, as circular E-RD cycles having length greater than 16 are very hard to detect and even harder to analyze for discovering associated autofunction constraints [19]) and $|I|$ is the average number of implied constraints by a non-relational autofunction constraint (generally between 0 and 5 [16]).

Moreover, these algorithms are, in fact, also optimal, as they search in every object-oriented class only within the event-driven methods and no such method is visited twice: their implementations merge for updates in $A1$ and deletions in $A2$ code injections and deletions in a same step, whereas checking of implied constraints by a newly added one in $A2$ is done in internal memory (and are saved in the db together with the one done by users in c's check-box, when the current row corresponding to f from *FUNCTIONS* is saved).

In fact, $A1$ is not implemented as such in *MatBase*, but split into the event-driven methods *OnDelete* (for delete requests) and *BeforeUpdate* / *Validating* for the control *ObjectConstraint*

(for insert and update requests) of the form *OBJECT_CONSTRAINTS*. *A*2 is implemented in the *BeforeUpdate* / *Validating* for check-box controls associated to the non-relational constraint types of the form *FUNCTIONS*.

6.2 Algorithms Usefulness

First, analyzing non-relational constraints is interesting even *per se*, within the study of sets, functions, and relations semi-naïve algebra, as well as the one of data modeling and db constraint theory, as it helps getting a better understanding on both single and compound autofunctions. Moreover, the object-type constraints are excellent real-world examples of closed first order predicate calculus with equality formulas.

The main utility of these algorithms is, of course, in the realms of data modeling, db constraints theory, db and db software applications design and development. All constraints (business rules) that are governing the sub-universes modeled by dbs, be them relational or not, should be declared in the corresponding mathematical dbs' schemas and then enforced in its implementations: otherwise, their instances might be implausible.

Generally, almost all autofunctions have associated non-relational constraints [16]. For example, in the *MatBase* Geography db there are 7 compound autofunctions that are reflexive, 8 asymmetric, 7 irreflexive, one acyclic, and 16 have associated object constraints [16].

7. CONCLUSION AND FURTHER WORK

In summary, we have designed, implemented, and successfully tested in both *MatBase* current versions (for MS Access and C# and SQL Server, respectively) algorithms for automatic enforcement of the non-relational constraints associated to autofunctions, analyzed their complexity and optimality, as well as outlined their usefulness for both sets, functions, and relations algebra, and, especially, for data modelling, db constraints theory, db and db software application design and development practices.

Unfortunately, except for *MatBase* and BRMS users, the state of the art in db and db software application design and development related to non-relational constraints is exclusively using ad-hoc approaches: very few db and/or software architects are aware of their types, optimal enforcing ways per type, algorithmic approaches to discover all of them for any sub-universe of discourse, automatic enforcement, and even of their paramount importance; only from experience and common sense (e.g. nobody may be simultaneously present in several places, no slot may be simultaneously occupied by several objects, etc.) are they considering some such constraints and enforce them in db software applications (through either RDBMS triggers in extended SQL or/and high-level programming languages embedding SQL trigger-type methods).

Very many such constraints are only discovered in production, in time, generally by db software application users, who report them as bugs. It would be highly preferable, of course, that db and software architects discover all of them in the data modeling phase and then developers enforce them from the beginning, up until *MatBase* (or/and other similar advanced DBMS products) will become worldwide available, to provide automatic code generation for enforcing non-relational constraints too, just like it is the case today with the relational ones.

The algorithms presented in this paper 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), as well as by two Romanian IT companies developing db software applications for many U.S. and European customers in the Fortune 100

ones, whose productivities have greatly increased ever since, as lot of software designing, developing, testing, and debugging efforts are spared by *MatBase* automatic code generation.

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

Further work needs to be done for also assisting users in guaranteeing coherence and minimality for object-type constraints. This is not at all an easy task, as even the problem for the subclass of first order calculus predicate closed formulas is NP-complete.

REFERENCES

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

[2] Agiloft Reference Manual, https://www.agiloft.com/documentation/agiloft-reference-manual.pdf, last accessed 2019/03/28.

[3] Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based Scheduling. Kluwer Academic Publishers, Dordrecht, NL (2001).

[4] Chen, P. P.: The entity-relationship model: Toward a unified view of data. ACM Transactions on Database Systems 1(1), 9–36 (1976).

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

[6] Dyer, L., et all.: Scaling BPM Adoption from Project to Program with IBM Business Process Manager. ibm.com/redbooks,http://www.redbooks.ibm.com/redbooks/pdfs/sg247973.pdf, last accessed 2019/03/28.

[7] Fourer, R., Gay, D. M.: Extending an Algebraic Modeling Language to Support Constraint Programming. INFORMS Journal on Computing, 14 (4), 322–344 (2002).

[8] von Halle, B.: Business Rules Applied: Building Better Systems Using the Business Rules Approach. John Wiley and sons, New-York, NY (2001).

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

[10] Kolban, N.: Kolban's Book on IBM Decision Server Insights. ibm.com/redbooks,http://neilkolban.com/ibm/wp-content/uploads/2015/06/Kolbans-ODM-DSI-Book-2015-06.pdf, last accessed 2019/03/28.

[11] Krzysztof, A.: Principles of Constraint Programming. Cambridge University Press, Cambridge, UK (2003).

[12] Mancas, C.: On Modeling Closed E-R Diagrams Using an Elementary Mathematical Data Model. In: Proc. 6th ADBIS Conference on Advances in DB and Inf. Syst., pp. 65-74. Slovak Technology University Press, Bratislava, Slovakia (2002).

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

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

[15] 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 (2020, in press).

[16] Mancas, C.: *MatBase* E-RD Cycles Associated Non-Relational Constraints Discovery Assistance Algorithm. In: Intelligent Computing, Proc. 2019 Computing Conference, AISC Series 997, vol.1, pp. 390 – 409, Springer Nature, Switzerland (2019).

[17] Mancas, C.: *MatBase* – a Tool for Transparent Programming while Modelling Data at Conceptual Levels, In: Proc. 5th Intl. Conf. on Computer Science, Information Technology (CSITEC 2019), pp. 15 – 27, AIRCC Pub. Corp., India (2019).

[18] Mancas, C., Dorobantu, V.: On Enforcing Relational Constraints in *MatBase*. London Journal of Research in Comp. Sci. and Technology 17, 1 (Jan. 2017), 39-45 (2017).

[19] Mancas, C., Mocanu, A.: *MatBase* DFS Detecting and Classifying E-RD Cycles Algorithm. Journal of Computer Science Applications and Information Technology2 (4), 1–14(2017).

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

[21] Red Hat Customer Content Services: Getting Started with Red Hat Business Optimizer. https://access.redhat.com/documentation/en-us/red_hat_decision_manager/7.1/pdf/getting_started_with_red_hat_business_optimizer/Red_Hat_Decision_Manager-7.1-Getting_started_with_Red_Hat_Business_Optimizer-en-US.pdf, last accessed 2019/03/28.

[22] Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. Logos Verlag, Berlin, Germany (2001).

[23] Rina, D.: Constraint processing. Morgan Kaufmann Publishers, Boston, MA (2003).

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

[25] Thalheim, B.: Entity-Relationship Modeling: Foundations of Database Technologies. Springer-Verlag, Berlin, Germany (2000).

[26] Thom, F., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag, Berlin, Germany (2003).

[27] Weiden, M., Hermans, L., Schreiber, G., van der Zee, S.: Classification and Representation of Business Rules. In: Proc. 2002 European Business Rules Conference, http://www.omg.org/docs/ad/02-12-18.pdf, last accessed 2010/07/02.

AUTHOR

Christian Mancas has graduated in 1977 the Computers Department of the *Politehnica* University of Bucharest, Romania. He obtained his PhD degree in 1997 from the same as above Department. He first worked as a software engineer and, since 1980, R&D manager of a Computer Centre in Bucharest. Since 1990, he worked for several IT start-ups, including his owns, as data architect, software infrastructure manager, etc. Since 1998, he is an Associate Professor with both the Mathematics and Computer Science Department of the *Ovidius* University, Constanta, and (as an invited Professor) Engineering Taught in Foreign Languages Department (Computer Science and Telecommunications in English stream) of the *Politehnica* University, Bucharest, Romania.