A Frame Manipulation Algebra for ER Logical Stage Modelling

Antonio L. Furtado, Marco A. Casanova, Karin K. Breitman, Simone D. J. Barbosa

Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de S. Vicente, 225. Rio de Janeiro, RJ. Brasil - CEP 22451-900
{furtado, casanova, karin, simone}@inf.puc-rio.br

Abstract. The ER model is arguably today's most widely accepted basis for the conceptual specification of information systems. A further common practice is to use the Relational Model at an intermediate logical stage, in order to adequately prepare for physical implementation. Although the Relational Model still works well in contexts relying on standard databases, it imposes certain restrictions, not inherent in ER specifications, which make it less suitable in Web environments. This paper proposes frames as an alternative to move from ER specifications to logical stage modelling, and treats frames as an abstract data type equipped with a Frame Manipulation Algebra (FMA). It is argued that frames, with a long tradition in AI applications, are able to accommodate the irregularities of semi-structured data, and that frame-sets generalize relational tables, allowing to drop the strict homogeneity requirement. A prototype logic-programming tool has been developed to experiment with FMA. Examples are included to help describe the use of the operators.

Keywords. Frames, semi-structured data, abstract data types, algebra.

1 Introduction

It is widely recognized [29] that database design comprises three successive stages:

a. conceptual,
b. logical,
c. physical.

The Entity-Relationship (ER) model has gained ample acceptance for stage (a), while the Relational Model is still the most popular for (b) [29]. Stage (c) has to do with implementation using some DBMS compatible with the model chosen at stage (b).

Design should normally proceed top-down, from (a) to (b) and then to (c). Curiously the two models mentioned above were conceived, so to speak, in a bottom-up fashion. The central notion of the Relational Model – the relation or table – corresponds to an abstraction of conventional file structures. On the other hand, the originally declared purpose of the ER model was to subsume, and thereby conciliate, the Relational Model and its competitors: the Hierarchic and the Codasyl models [9].

Fortunately, the database research community did not take much time to detect the radical distinction between the ER model and the other models, realizing that only the former addresses *conceptual modelling*, whereas the others play their part at the stage of *logical modelling*, as an intermediate step along the often worksome passage from world concepts to machine implementation. To that end, they resort to different data structures (respectively: tables, trees, networks). Tables in particular, once equipped with a formal language for their manipulation – namely Relational Algebra or Relational Calculus [12] – constitute a full-fledged *abstract data type*.

Despite certain criticisms, such as the claim that different structures might lead to a better performance for certain modern business applications [28], the Relational Model still underlies the architecture of most DBMSs currently working on conventional databases, some of which with an extended object-relational data model to respond to the demand for object-oriented features [3,29]. However, in the context of Web environments, information may come from a variety of sources, in different formats, with little or no structure, and is often incomplete or conflicting. Moreover the traditional notion of classification as conformity to postulated lists of properties has been questioned [21], suggesting that similarity to typical representatives might provide a better criterion, as we investigated [1] employing a three-factor measure.

We suggest that *frames*, with a long tradition in Artificial Intelligence applications [4,22], provide an adequate degree of flexibility. The main contribution of the present paper is to propose a Frame Manipulation Algebra (FMA) to fully characterize frames and frame-sets as an abstract data type, powerful enough to help moving from the ER specifications to the logical design stage.

The paper is organized as follows. Section 2 recalls how facts are characterized in the ER model, and describes the clausal notation adopted for their representation. In section 3, four kinds of relations between facts are examined, as providing a guiding criterion to choose a (in a practical sense) *complete* repertoire of operators for manipulating information-bearing structures, such as frames. Section 4, which is the thrust of the paper, discusses frames, frame-sets and the FMA operators, together with extensions that enhance their application. Section 5 contains concluding remarks.

2 Facts in terms of the ER model

A database *state* consists of all *facts* that hold in the mini-world underlying an information system at a certain moment of time. For the sake of the present discussion, we assume that all incoming information is first broken down into basic facts, represented in a standard unit clause format, in full conformity with the ER model. We also assume that, besides facts, meta-level conceptual schema information is represented, also in clausal format.

Following the ER model, facts refer to the existence of entity *instances* and to their *properties*. These include their attributes and respective values and their participation in binary relationships, whose instances may in turn have attributes. Schema information serves to characterize the allowed *classes* of entity and relationship instances. Entity classes may be connected by is_a and part_of links. A notation in a logic programming style is used, as shown below (note that the identifying attribute of an entity class is indicated as a second parameter in the `entity` clause itself):

```
Schema
entity(<entity name>,<identifying attribute>)
attribute(<entity name>,<attribute name>)
domain(<entity name>,<attribute name>,<value domain specification>)
relationship(<relationship name>,[<entity name>,<entity name>])
attribute(<relationship name>,<attribute name>)
is_a(<entity name>,<entity name>)
part_of(<entity name>,<entity name>)

Instances
<entity name>(<identifier value>)
<attribute name>(<identifier value>,<attribute value>)
<relationship name>([<identifier value>,<identifier value>])
<relationship attribute>([<identifier value>,<identifier
value>],<attribute value>)
```

For entities that are *part-of* others, `<identifier value>` is a list of identifiers at successive levels, in descending order. For instance, if companies are downward structured in departments, sections, etc., an instance of a `quality control` section might be designated as `section(['Acme', product, quality_control])`.

A common practice is to *reify* n-ary relationships, for n > 2, i.e. to represent their occurrence by instances of appropriately named entity classes. For example, a `ships` ternary relationship, between entity classes `company`, `product` and `client`, would lead to an entity class `shipment`, connected to the respective participating entities by different binary relationships, such as `ships_agent`, `ships_object`, `ships_recipient` to use a case grammar nomenclature [16]. To avoid cluttering the presentation with details, such extensions and other notational features will not be covered here, with two exceptions to be illustrated in examples 3 and 8 (section 4.3). Also not covered are non-conventional value domains, e.g. for multimedia applications, which may require an extensible data type feature [27].

The clausal notation is also compatible with the notation of the RDF (*Resource Description Framewok*) language. A correspondence may be established between our clauses and RDF statements, which are triples of the form `(<subject>, <property or predicate>, <object>)` [6], if we replace `<subject>` by `<identifier value>`. It is worth noting that RDF has been declared to be "a member of the Entity-Relationship modelling family" in *The Cambridge Communiqué*, a W3C document[1].

3 Relations between facts

Facts should be articulated in a *coherent* way to form a meaningful utterance. Starting from semiotic studies [5,7,8,24], we have detected four types of relations between facts – *syntagmatic*, *paradigmatic*, *antithetic*, and *meronymic* – referring, respectively, to *coherence* inside an utterance, to *alternatives* around some common paradigm, to negative *restrictions*, and to successive *levels of detail*.

Such relations serve to define the dimensions and limits of the information space, wherein facts are articulated to compose meaningful utterances, which we represent at the logical stage as frames, either standing alone or assembled in frame-sets. In turn, as will be shown in section 4.2, the characterization of the relations offers a criterion to configure an adequate repertoire of operators to handle frames and frame-sets.

[1] www.w3.org/TR/schema-arch

3.1 Syntagmatic relations

Adapting a notion taken from linguistic studies [24], we say that a *syntagmatic relation* holds between facts F1 and F2 if they express properties of the same entity instance Ei. Since properties include relationships in which the entity instance participates, the syntagmatic relation applies transitively to facts pertaining to other entity instances connected to Ei via some relationship. The syntagmatic relation acts therefore as a fundamental reason to chain different facts in a single *cohesive* utterance. For example, it would be meaningful to expand John's frame by joining it to the `headquarters` property belonging to the frame of the `company` he works for.

On the other hand, if an entity instance has properties from more than one class, an utterance may either encompass all properties or be restricted to those of a chosen class. For example, if John is both a student and an employee, one might be interested to focus on properties of John as a `student`, in which case his `salary` and `works` properties would have a weaker justification for inclusion.

3.2 Paradigmatic relations

Still adapting [24], a *paradigmatic relation* holds between facts F1 and F2 if they constitute *alternatives* according to some criterion (paradigm). The presence of this relation is what leads to the formation of frame-sets. To begin with, all facts involving the same property are so related, such as John's salary and Mary's salary. Indeed, since they are both `employees`, possibly sharing additional properties, a frame-set including their frames would make sense, recalling that the most obvious reason to create conventional files is to gather all data pertaining to instances of an entity class.

Property similarity is still another reason for a paradigmatic relation. For example, `salary` and `scholarship` are similar in that they are alternative forms of income, which would justify assembling `employees` and `students` in one frame-set with the purpose of examining the financial status of a population group.

Even more heterogeneous frame-sets may arise if the unifying paradigm serves an occasional pragmatic objective, such as to provide all kinds of information of interest to a trip, including flight, hotel and restaurant information. A common property, e.g. `city`, would then serve to select whatever refers to the place currently being visited.

3.3 Antithetic relations

Taken together, the syntagmatic and paradigmatic relations allow configuring two dimensions in the information space. They can be described as orthogonal, if, on the one hand, we visualize the "horizontal" syntagmatic axis as the one along which frames are created by aligning properties and by the concatenation with other frames or subsequences thereof, and, on the other hand, the "vertical" paradigmatic axis as the one down which frames offering alternatives within some common paradigm are assembled to compose frame-sets.

And yet orthogonality, in the specific sense of independence of the two dimensions, sometimes breaks down due to the existence of antithetic relations. An *antithetic relation* holds between two facts if they are *incompatible* with each other. Full orthogonality would imply that a fact F1 should be able to coexist in a frame with any alternative facts $F2_1$, ..., $F2_n$ characterized by the same paradigm, but this is not

so. Suppose we are told that Mary is seven years old; then she can have `scholarship` as income, but not `salary`, if the legislation duly restricts the `age` for employment.

Thus antithetic relations do not introduce a new dimension, serving instead to delimit the information space. Suggested by semiotic research on binary oppositions and irony [5,7], they are the result of *negative prescriptions* from various origins, such as natural impossibilities, laws and regulations, business rules, integrity constraints, and any sort of decisions, justifiable or arbitrary. They may motivate the absence of some property from a frame, or the exclusion of one or more frames from a frame-set. For example, one may want to exclude the recent graduates from a students frame-set. Ironically, such restrictions, even when necessary for legal or administrative reasons, may fail to occur in practice, which would then constitute cases of violation or, sometimes, of admissible exceptions.

3.4 Meronymic relations

Meronymy is a word of Greek origin, used in linguistics to refer to the decomposition of a whole into its constituent parts. Forming an adjective from this noun, we shall call *meronymic relations* those that hold between a fact F1 and a lower-level set of facts $F2_1$, $F2_2$, ..., $F2_n$, with whose help it is possible to achieve more *detailed* descriptions. The number of levels may of course be greater than two.

The correspondence between a fact, say F1, with a lower-level set of facts $F2_1$, $F2_2$, ..., $F2_n$ requires, in general, some sort of mapping rule. Here we shall concentrate on the simplest cases of decomposition, where the mapping connections can be expressed by *part-of* semantic links of the component/ integral-object type (cf. [31]). A `company` may be subdivided into `departments`, which may in turn have `sections` and so on and so forth. A `country` may have `states`, `townships`, etc. Outside our present scope is, for instance, the case of `artifacts` whose `parts` are interconnected in ways that could only be described through maps with the descriptive power of a blueprint.

Meronymic relations add a third dimension to the information space. If discrete levels of detail are specified, we can visualize successive two-dimensional planes disposed along the meronymic axis, each plane determined by its syntagmatic and paradigmatic axes.

Traversing the meronymic axis is like zooming in or out. After looking at a `company` frame, one may want to come closer in order to examine the frames of its constituent `departments`, and further down towards the smallest organizational units, the same applying in turn to each frame in a frame-set describing several `companies`. And while the *is-a* links imply top-down property inheritance, *part-of* links induce a bottom-up aggregation of values. For example, if there is a `budget` attribute for each `department` of a `company`, summing up their values would yield a corporate total.

4 Towards an abstract data type for ER logical-stage modelling

4.1 Frames and frame-sets

Frames are sets of P:V (i.e. `<property>:<value>`) pairs. A frame-set can either be the empty set [] or consist of one or more frames.

The most elementary frames are those collecting P:V information about a single entity or binary relationship instance, or a single class. In a frame displaying information on a given entity instance E, each property may refer to an attribute or to a relationship. In the latter case, the P component takes the form R/1 or R/2 to indicate whether E is the first or the second entity participating in relationship R, whereas the V component is the identifier (or list of identifiers) of the entity instance (or instances) related to E by R. In a frame displaying information about a relationship instance, only attributes are allowed as properties. For frames concerning entity or relationship classes, the V component positions can be filled up with variables.

We require that a property cannot figure more than once in a frame, a restriction that has an important consequence when frames are compared during the execution of an operation: by first sorting each frame, i.e. by putting the P:V pairs in lexicographic order (an `n×log(n)` process), we ensure that the comparisons proper take linear time.

A few examples of elementary frames follow. The notation "_" indicates an anonymous variable. Typically not all properties specified for a class will have known values for all instances of the class. If, among other properties, Mary's `age` is unknown at the moment, this information is simply not present in her frame. The last line below illustrates a frame-set, whose constituent frames provide information about two employees of company Acme.

Class employee: `[name:_, age:_, salary:_, works/1:_]`
Class works: `[name:_,cname:_,status:_}`
Mary: `[name:'Mary', salary:150, works/1:'Acme']`
John:
```
[name: 'John', age: 46, salary: 100, scholarship: 50,
 works/1: 'Acme']
```
Acme: `[cname:'Acme', headquarters:'Carfax', works/2:['John','Mary']]`
Acme employees:
```
[ [name:'Mary', salary:150, works/1:'Acme'],
  [name: 'John', age: 46, salary: 100, scholarship: 50,
   works/1: 'Acme'] ]
```

Both Acme's frame and Mary's frame contain, respectively, properties of a single class or instance. However, if we want that frames should constitute a realistic model for human utterances, more complex frames are needed. In particular, the addition of properties of related identifiers should be allowed, as in:

```
[name: 'Mary', salary: 150, works/1: 'Acme', headquarters: 'Carfax',
 status: temporary, 'John'\salary: 100]
```

where the third property belongs to the company for which Mary works, and the fifth is a relationship attribute concerning her job at the company. The inclusion of the sixth property, which belongs to her co-worker John, would violate the syntactic requirement that property names be unique inside a frame; the problem is solved by prefixing the other employee's salary property with his identifier. Further generalizing this practice, for the sake of clarity, one may choose to fully prefix in this way all properties attached to identifiers other than Mary:

```
[name:'Mary', salary:150, works/1:'Acme',
 'Acme'\headquarters:'Carfax', ['Mary','Acme']\status: temporary,
 'John'\salary: 100]
```

Recalling that every instance is distinguished by its `<identifier value>`, we may establish a correspondence between an instance frame and a labelled RDF-graph whose edges represent triples sharing the same `<subject>` root node [6].

4.2 Overview of the algebra

Both frames and frame-sets can figure in FMA expressions as operands. To denote the evaluation of an expression, and the assignment of the resulting frame or frame-set to a variable `F`, one can write:

```
F := <algebraic expression>.
```

or optionally:

```
F#r := <algebraic expression>.
```

in which case, as a side-effect, the expression itself will be stored for future use, the indicated `r` constant serving thereafter as an identifier. Storing the result rather than the expression requires two consecutive steps:

```
F1 := <algebraic expression>
F2#r := F1.
```

A stored expression works like a database *view*, since every time the expression is evaluated, the result will vary according to the current state, whereas storing a given result corresponds to a *snapshot*.

The simplest expressions consist of a single frame, which may be represented explicitly or by an instance identifier (or `r` constant) or class name, in which case the FMA engine will retrieve the respective properties to compose the result frame. Note that the first and the second evaluations below should yield the same result, whereas the third yields a frame limited to the properties specified in the *search-frame* placed after the `^` symbol (example 11 shows a useful application of this feature). If the "`\`" symbol is used instead of "`^`", the full-prefix notation will be applied. Note, in addition, that lists of identifiers or of class names yield frame-sets.

```
Fm1 := [name:'Mary', salary:150, works/1:'Acme']
Fm2 := 'Mary'.
Fms1 := 'Mary' ^ [salary:S, works/1:C].
Fmsp := 'Mary' \ [salary:S, works/1:C].
Fmsr#msw := 'Mary' \ [salary:S, works/1:C].
Fms2 := msw.
Fmj1 := [[name:'Mary', salary:150, works/1:'Acme'], [name:'John', age:46,
         salary:100, scholarship:50, works/1:'Acme']]
Fmj2 := ['Mary','John'].
Fc := student.
```

Instances and classes can be treated together in a particularly convenient way. If John is both a student and an employee, his properties can be collected in separate frames, by indicating the name of each class, whose frame will then serve as search-frame:

```
Fjs := 'John' ^ student.
Fje := 'John' ^ employee.
```

Over these simple terms, the *algebraic operators* can be used to build more complex expressions. To build the operator set of FMA, the five basic operators of Relational Algebra were redefined to handle both frames and frame-sets. Two more operators had to be added in order to take into due account all the four relations between facts indicated in section 3.

An intuitive understanding of the role played by the first four operators is suggested when they are grouped into pairs, the first operator providing a *constructor* and the second a *selector*. This is reminiscent of the LISP primitives, where `cons` works as constructor and `car` and `cdr` as selectors, noting that `eq`, the primitive on which value comparisons ultimately depend, induces yet another selector mechanism. For FMA the two pairs are:

- *product* and *projection*, along the syntagmatic axis;
- *union* and *selection*, along the paradigmatic axis.

Apart from constructors and selectors, a negation operator is needed, as demanded by antithetic restrictions. To this end, FMA has the *difference* operator and enables the selection operator to evaluate logical expressions involving the *not* Boolean operator. LISP includes *not* as a primitive, and Relational Algebra has difference. Negation is also essential for expressing universal in terms of existential quantification. Recall for example that a `supplier` who `supplies` all `products` is anyone such that there is *not* some `product` that it does *not* `supply`. Also, difference being provided, an intersection operator is no longer needed as a primitive, since $A \cap B = A - (A - B)$.

To traverse the meronymic dimension, zooming in and out along part-of links, FMA includes the *factoring* and the *combination* operators. One must recall at this point that the Relational Model originally required that tables be in first-normal form (1NF), which determined the choice of the Relational Algebra operators and their definition, allowing only such tables as operands. However, more complex types of data, describing for example assembled products or geographical units, characterized conceptually via a semantic *part-of* hierarchy [26], led to the use of the so-called NF^2 (non first normal form) or nested tables at the logical level of design. To handle NF^2 tables, an extended relational algebra was needed, including operators such as "partitioning" and "de-partitioning" [18], or "nest" and "unnest" [19] to convert from 1NF into NF^2 tables and vice-versa.

We claim that, with the seven operators indicated here, FMA is complete in the specific sense that it covers frame (and frame-set) manipulation in the information space spanned by the syntagmatic, paradigmatic, antithetic and meronymic relations holding between facts.

It has been demonstrated that Relational Algebra is *complete*, in that its five operators are enough, as long as only 1NF tables are permitted, to make it equivalent in expressive power to Relational Calculus, a formalism based on first-order calculus. Another aspect of completeness is *computational completeness* [14,30], usually measured through a comparison with a Turing machine. To increase the computational power of relational DBMSs, the SQL-99 standard includes provision for recursive queries.

Pursuing along this trend, we decided to embed our running FMA prototype in a logic programming language, which not only made it easier to define virtual attributes and relationships, a rather flexible selection operator and an iteration extension, but also to take advantage of Prolog's pattern-matching facilities to deal simultaneously with instance frames and (non-ground) frame patterns and class frames.

4.3 The basic Algebraic operators

Out of the seven FMA operators, three are binary and the others are unary. All operators admit both frames and frame-sets as operands. For union, selection and difference, if frames are given as operands, the prototype tool transforms them into frames-sets as a preliminary step; conversely, the result will be converted into frame format whenever it is a frame-set containing just one frame.

Apart from this, the main differences between the way that FMA and the Relational Algebra treat the five operators that they have in common are due to the relaxation of the homogeneity and first-normal form requirements. In Relational Algebra, union and difference can only be performed on union-compatible tables. Since union-compatibility is not prescribed in FMA, the frames belonging to a frame-set need not be constituted of exactly the same properties, which in turn affects the functioning of the projection and selection operators. Both operators search for a number of properties in the operand, but no error is signaled if some property is missing in one or more frames: such frames simply do not contribute to the result. FMA also differs from Relational Algebra by permitting arbitrary logical expressions to be tested as an optional part of the execution of the selection operator. Moreover the several uses of variables, enabled by logic programming, open a number of possibilities, some of which are illustrated in the examples.

The empty list "[]" (nil) is used to ambiguously denote the empty frame and the empty frame-set. As such, [] works as the neutral element for both product and union and, in addition, is returned as the result when the execution of an operator fails, for example when no frame in a frame-set satisfies a selection test.

The FMA requirement that a property can occur at most once in a frame raises a conflict if, when a product is executed, the same property figures in both operands. The conflict may be solved by default if the attached values are the same, or may require a decision, which may be fixed beforehand through the inclusion of appropriate tags. Handling conflicts through the use of tags is a convenient expedient that serves various purposes, such as to replace a value, or form sets or bags (recalling that multiple values are permitted), or call for aggregate numerical computations, etc. If no tag is supplied, our prototype tool offers a menu to the user's choice.

The two operators without counterpart in Relational Algebra, namely factoring and combination, act on frame-structured identifiers associated with *part-of* links, and also on attributes with frame-structured value domains. When working on a list of identifiers, the result of factoring is a frame-set composed of the frames obtained from each identifier in the operand list. When working on properties with frame-structured value domains, factoring has a flattening effect, breaking the property into separate constituents so as to bring to the front the internal structure.

When examining the examples, recall that, although the operands of every FMA operation are always frames or frame-sets, identifiers or lists of identifiers may figure in their place, being converted into the corresponding frames or frame-sets as a preliminary step in the execution of the operation. Both in the description of the operators and in the examples, we shall employ a notation that is unavoidably a transliteration imposed by the Prolog character set limitations and syntax restrictions. For instance, "+" denotes union. Also, since blank spaces are not allowed as separators, the operand of a projection or selection is introduced by an "@" symbol.

Product. The product of two frames F1 and F2, denoted F1 * F2, returns a frame F containing all F1 and F2 properties. If one or both operands are (non-empty) frame-sets, the result is a frame-set containing the product of each frame taken from the first operand with each frame from the second, according to the standard Cartesian product conventions. If one of the operands is the empty frame, denoted by [], the result of the product operation is the other operand, and thus [] behaves as the neutral element for product. The case of an empty frame-set, rather than an empty frame, demanded an implementation decision; by analogy with the zero element in the algebra of numbers, it would be justifiable to determine that a failure should result whenever one or both operands are an empty frame-set. However we preferred, here again, to return the other operand as result, so as to regard the two cases (i.e. product by empty frame or by empty frame-set) as frustrated attempts to extend frames, rather than errors.

When two operand frames have one or more properties in common, a conflict arises, since, being a frame, the result could have no more than one P:V pair for each property P. Except if V is the same in both operands, the criterion to solve the conflict must be indicated explicitly through a P:τ(V) notation, where, depending on the choice of the tag τ, the values V1 and V2 coming from the two operands can be handled as follows to obtain the resulting V, noting that one or both can be value lists:

- $\tau \in$ {`set`,`bag`} – V is a set or bag (which keeps duplicates and preserves the order), containing the value or values of property P taken from V1 and V2;
- $\tau =$ `del` – V is the set difference V1 - V2, containing therefore the value or values in V1 not also present in V2;
- $\tau =$ `rep` – V is V2, where V2 is either given explicitly, or results from an expression indicating the replacement of V1 by V2 (cf. example 1);
- $\tau \in$ {`sum`,`min`,`max`,`count`,`avg`} – V is an aggregate value (cf. section 4.4, example 9).

A more radical effect is the removal of property P, so that no pair P:V will appear in the result, which happens if one operand has P:`nil`.

Notice, finally, that the conflict may be avoided altogether by adding a suitable prefix to the occurrence in one or both operands, as in S1\P:V1 and/or S2\P:V2, in which case the two occurrences will appear as distinct properties in the result.

Example 1: Suppose that one wishes to modify the values of the salary attribute of a group of employees, say John and Mary, figuring in a frame-set, by granting a 5% raise. This can be done by specifying a frame containing a replacement tag and then performing the product of this frame against the given frame-set. In the replacement tag shown in the first line, X refers to the current salary and Y to the new salary, to be obtained by multiplying X by 1.05 (note that ":-" is the prompt for Prolog evaluation):

```
:- F := [salary:rep(X/(Y:(Y is X * 1.05)))] *
        [[name:'John',salary:130], [name:'Mary',salary:150]].
```

result: `F = [[name:John, salary:136.50], [name:Mary, salary:157.50]]`

Projection. The projection of a frame F', denoted proj [T] @ F', returns a frame F that only contains the properties of F' specified in the projection-template T, ordered according to their position in T. The projection-template T is a sequence of property names P or, optionally, of P:V pairs, where V is a value in the domain of property P or is a variable. In addition to (or instead of) retrieving the desired properties, projection can be used to display them in an arbitrary order. Note that, for efficiency,

all operations preliminarily sort their operands and, as a consequence – with the sole exception of projection, as just mentioned – yield their result in lexicographic order.

If the operand is a frame-set, the result is a frame-set containing the projection of the frames of the operand. Note however that, being sets, they cannot contain duplicates, which may arise as the consequence of a projection that suppresses all the property-value pairs that distinguish two or more frames – and such duplicates are accordingly eliminated from the result. If the projection fails for some reason, e.g. because the projection-template T referred to a P or P:V term that did not figure in F', the result will be [] rather than an error.

Example 2: Product is used to concatenate information belonging to Mary's frame with information about the company she works for, and with an attribute pertaining to her work relationship. Projection is used to display the result in a chosen order.

```
:- F1 := 'Mary' ^ [name:N,works/1:C] *
         C ^ [headquarters:H] *
         works(['Mary',C]) ^ [status:S],
   F2 := proj [name,status,works/1,headquarters] @ F1.
```

result: `F = [name:Mary, status:temporary, works/1:Acme,headquarters:Carfax]`

Example 3: Given a list of identifiers, their frames are obtained and the resulting frame-set assigned to F1. Projection on `name` and `revenue` fails for Dupin. Notice that `revenue` has been defined as a virtual attribute, a sum of `salary` and `scholarship`.

```
revenue(A, D) :-
        bagof(B, (salary(A, B);scholarship(A, B)), C),
        sum(C, D).

:- F1 := ['Mina','Dupin','Hercule'],
   F2 := proj [name,revenue] @ F1.
```

result: `F = [[name:Mina, revenue:50], [name:Hercule, revenue:130]]`

Union. The union of two frames F1 and F2, denoted by F1 + F2, returns a frame-set containing both F1 and F2. If one or both operands are frame-sets, the result is a frame-set containing all frames in each operand, with duplicates eliminated. One or both operands can be the empty frame-set, ambiguously denoted as said before by [], functioning as the neutral element for union; so, if one of the operands is [], the union operator returns the other operand as result. In all cases, resulting frame-sets consisting of just one frame are converted into single frame format.

Example 4: The common paradigm, leading to put together hotel and airport-transfer frames, is the practical need to assemble any information relevant to a trip. The resulting frame-set is assigned to F and also stored under the `my_trip` identifier.

```
:- F#my_trip := [[hotel: 'Bavária',city: 'Gramado'],
                 [hotel: 'Everest',city: 'Rio']] +
                [transfer_type: executive,airport:'Salgado Filho',
                 to: 'Gramado', departure: '10 AM'].
```

result:
```
F = [[hotel: 'Bavária',city: 'Gramado'],
     [hotel: 'Everest',city: 'Rio'],
     [transfer_type: executive,airport:'Salgado Filho',
      to: 'Gramado', departure: '10 AM']]
```

Selection. The selection of a frame F', denoted sel [T]/E @ F', returns the frame F' itself if the selection-template T matches F', and the subsequent evaluation of the selection-condition E (also involving information taken from F') succeeds. The presence of E is optional, except if T is empty. If the test fails, the result to be assigned to F is the empty frame []. If the operand is a frame-set, its result will be a

frame-set containing all frames that satisfy the test, or the empty frame-set [] if none does. Resulting frame-sets consisting of just one frame are converted into frame format. In order to select one plot at a time from a resulting frame-set S containing two or more frames, the form sel [T]/E @ one(S) must be employed.

Example 5: Since `my_trip` denotes a previously computed and stored frame-set (cf. example 4), it is now possible to select from `my_trip` all the information concerning Gramado, no matter which property may have as value the name of this city (notice the use of an anonymous variable in the selection-template). The result is stored, under the `er_venue` identifier.

```
:- F#er_venue := sel [_: 'Gramado'] @ my_trip.
```

result:
```
F = [[airport: Salgado Filho, time: 10 AM, to: Gramado,
      transfer_type: executive],
     [city: Gramado, hotel: Bavária]]
```

Difference. The difference of two frames F1 and F2, denoted F1 – F2, returns [] if F1 is equal to F2, or F1 otherwise. If one or both operands are frame-sets, the result is a frame-set containing all frames in the first operand that are not equal to any frame in the second. Resulting frame-sets with just one frame are converted into frame format.

Example 6: Assume, in continuation to examples 4 and 5, that one is about to leave Gramado. Difference is then used to retrieve information for the rest of the trip.

```
:- F := my_trip - er_venue.
```

result:
```
F = [hotel: 'Everest', city: 'Rio']
```

Factoring. The factoring of a frame-structured identifier I' of an entity instance, denoted by fac I', is a frame-set I containing the frame-structured identifiers I1,I2,...,In of all entity instances to which I' is directly connected by a *part-of* link.

Factoring can also be applied to frames that include attributes with frame-structured values. If F' is one such frame, its factoring F := fac F' is the result of expanding F', i.e. all terms A:[A1:V1,A:2:V2,...,An:Vn] will be replaced by the sequence A_A1:V1, A_A2:V2,..., A_An:Vn.

In both cases, if the operand is a frame-set, the result is a frame-set containing the result obtained by factoring each constituent of the operand.

Example 7: Given a list of company identifiers, the frame-structured identifiers of their constituent departments are obtained through factoring.

```
:- F := fac ['Acme', 'Casa_Soft'].
```

result:
```
F = [[1:VL, 2:personnel], [1:VL, 2:product], [1:VL, 2:sales],
     [1:BK, 2:audit], [1:BK, 2:product]]
```

Combination. The combination of a frame-structured identifier I' of an entity instance, denoted by comb I', is the frame-structured identifier I of the entity instance such that I' is *part-of* I. If the operand is a frame-set composed of frame-structured identifiers (or frame-sets thereof, as those obtained by factoring in example 7), the result is a frame-set containing the combinations of each constituent frame. Since duplicates are eliminated, all frame-structured identifiers Ij1',Ij2',...,Ijn' in I' that are *part-of* the same entity instance Ij will be replaced by a single occurrence of Ij in the resulting frame-set I.

Combination can also be applied to a frame F' containing expanded terms. Then F := comb F' will revert all such terms to their frame-structured value representation. The operand can be a frame-set, in which case the resulting frame-set will contain the result of applying combination to each constituent of the operand.

Example 8: Applying combination to frame F1, containing Carrie Fisher's data in flat format, yields frame F2, where `address` and `birth_date` are shown as properties with frame-structured values. This only works, however, if the two attributes have been explicitly defined, with the appropriate syntax, over frame-structured domains.

```
attribute(person,address).
domain(star,address,[street,city]).

attribute(person,birth_date).
domain(person,birth_date,[day,month,year]).

:- F := comb [name: 'Carrie Fisher', address_city: 'Hollywood',
              address_street: '123 Maple St.', birth_date_day: 21,
              birth_date_month:10,birth_date_year: 56,
              starred_in/1:'Star Wars'].
```

result:
```
F =  [name:Carrie Fisher,starred_in/1:Star Wars,
      address:[street:123 Maple St., city:Hollywood],
      birth_date:[day:21, month:10, year:56]
```

4.4 Extensions

As a convenient enhancement to its computational power, FMA allows to *iterate* over the two basic constructors, product and union.

Given a frame F', the iterated product of F', expressed by F := prod E @ F', where E is a logical expression sharing at least one variable with F', is evaluated as follows:

- first, the iterator-template T is obtained, as the set of all current instantiations of E, and then:
- if T is the empty set, F = []
- else, if T = {t1, t2, ..., tn}, $F = F'_{t1} * F'_{\{t2, ..., tn\}}$

where F'_{ti} is the same as F' with its variables instantiated consistently with those figuring in ti, and letting the subscript in $F'_{\{ti+1, ..., tn\}}$ refer to the remaining instantiations of T to be used recursively at the next stages. As happens with (binary) product, this feature applies to single frames and to frame-sets.

Similarly, given a frame F', the iterated union of F', expressed by F := uni E @ F', where E is a logical expression sharing at least one variable with F', is thus evaluated:

- first, the iterator-template T is obtained, as the set of all current instantiations of E, and then:
- if T is the empty set, F = []
- else, if T = {t1, t2, ..., tn}, $F = F'_{t1} + F'_{\{t2, ..., tn\}}$

where F'_{ti} is the same as F' with its variables instantiated consistently with those figuring in ti, and letting the subscript in $F'_{\{ti+1, ..., tn\}}$ refer to the remaining instantiations of T to be used recursively at the next stages. Once again, as happens with (binary) union, this feature applies to single frames and to frame-sets.

Example 9: If departments have a `budget` attribute, we may wish to compute a total value for each company by adding the `budget` values of their constituent departments. Two nested iteration schemes are involved, with `uni` finding each company C, and `prod` iterating over the set SD of departments of C, obtained by applying the factoring operator to C. For all departments D which are members of SD, the corresponding `budget` values are retrieved and added-up, as determined by the `sum` tag in the selection-template, yielding the corporate `budget` values. Notice the use of

`C\` at the beginning of the second line, in order to prefix each value with the respective company name.

```
:- F := uni (company(C)) @
          C\(prod (SD := fac C, member(D,SD)) @
                  (sel [budget:sum(B)] @ D ^ [budget:B])).
```

result: `F = [[Acme\budget:60], [Casa_Soft\budget:20]]`

Example 10: The same constant can be used an arbitrary number of times to serve as an artificial identifier, which may provide a device with an effect similar to that of "tagging", in the sense that this word is used in the context of folksonomies [13]. Looking back at Example 4, suppose we have, along a period of time, collected a number of frames pertinent to the planned trip, and marked each of them with the same `my_trip` constant (cf. the notation F#r at the beginning of section 4.2). Later, when needed, the desired frame-set can be assembled by applying iterated union. Notice in this example the double use of variable T, first as iterator-template and then as operand. As iterator-template, T is obtained through the repeated evaluation of the expression `T := my_trip`, which assigns to T the set of all instances of `my_trip` frames, whose union then results in the desired frame-set F.

```
:- F#my_trip := [hotel: 'Bavária',city: 'Gramado'] ...
:- F#my_trip := [hotel: 'Everest',city: 'Rio'] ...
:- F#my_trip := [transfer_type: executive,airport:'Salgado Filho',
                 to: 'Gramado', departure: '10 AM'] ...
........
:- G := uni (T := my_trip) @ T.
```

result:
```
G = [[hotel: 'Bavária',city: 'Gramado'],
     [hotel: 'Everest',city: 'Rio'],
     [transfer_type: executive,airport:'Salgado Filho',
      to: 'Gramado', departure: '10 AM']]
```

Another extension has to do with the obtention of *patterns*, in special for handling class frames and instance frames simultaneously, and for similarity [15] rather than mere equality comparisons. Given a frame F, the pattern of F, denoted by patt F, is obtained from F by substituting variables for the values of the various properties.

Example 11: The objective is to find which employees are somehow similar to Hercule. Both in F1 and F2, the union iterator-template is obtained by evaluating all instances of the expression `employee(E), not E == 'Hercule', Fe := E`, which retrieves each currently existing employee name E, different from Hercule, and then obtains the frame Fe having E as identifier. The operand of both union operations is a product, whose second term is the more important. In F1, it is determined by the sub-expression `'Hercule' ^ Fe`, which looks for properties of Hercule using Fe as search-frame (see section 4.2). In F2, a weaker similarity requirement is used; the sub-expression `'Hercule' ^ (patt Fe)` produces the properties shared by the frames of Hercule and E with equal or different values, which are all displayed as variables thanks to a second application of `patt`. Finally, product is used to introduce `same_prop_val` or `same_prop` as new properties, in order to indicate who has been found similar to Hercule.

```
:- F1 := uni (employee(E), not E == 'Hercule', Fe := E) @
             ([same_prop_val:E] * 'Hercule' ^ Fe).
```

result:
```
F1 = [[same_prop_val: Jonathan, salary: 100, works/1: Acme],
      [same_prop_val: Mina, works/1:Acme]]
```

```
:- F2 := uni (employee(E), not E == 'Hercule', Fe := E) @
              ([same_prop:E] * (patt ('Hercule' ^ (patt Fe)))).
```

result:
```
F2 = [[same_prop: Jonathan, salary:_, works/1:_],
      [same_prop: Mina, salary:_, works/1:_],
      [same_prop: Hugo, salary:_, scholarship:_, works/1:_]]
```

5 Concluding remarks

We have submitted in the present paper that frames are a convenient abstract data type for representing heterogeneous incomplete information. We have also argued that, with its seven operators, our Frame Manipulation Algebra (FMA) is complete in the specific sense that it covers frame (and frame-set) manipulation in the information space induced by the syntagmatic, paradigmatic, antithetic and meronymic relations holding between facts. These relations, besides characterizing some basic aspects of frame handling, can be associated in turn, as we argued in [11], with the four major tropes (metonymy, metaphor, irony, and synecdoche) of semiotic research [5,8].

Frames aim at partial *descriptions* of the mini-world underlying an information system. In a separate paper [17], we showed how to use other frame-like structures, denominated *plots*, to register how the mini-world has evolved (cf. [10]), i.e. what *narratives* were observed to happen. Moreover we have been associating the notion of plots with plan-recognition and plan-generation, as a powerful mechanism to achieve *executable specifications* and, after actual implementation, *intelligent systems* that make ample use of online available meta-data originating from the conceptual modelling stage (comprising *static*, *dynamic* and *behavioural* schemas).

To business information systems we have added literary genres as domains of application of such methods. In fact, the *plot manipulation algebra* (PMA), which we developed in parallel with FMA in order to also characterize plots as abstract data types, proved to be applicable in the context of digital entertainment [20].

Another example of the pervasive use of frame or frame-like structures, in the area of Artificial Intelligence, is the seminal work on *stereotypes* [23] to represent personality traits. In the continuation of our project, we intend to pursue this line of research so as to enhance our behavioural characterization of *agents* (or *personages*, in literary genres), encompassing both cognitive and emotional factors [2].

References

1. Barbosa, S.D.J.; Breitman, K.K.; Furtado, A.L.; Casanova (2007). "Similarity and Analogy over Application Domains". Proc. *XXII Simpósio Brasileiro de Banco de Dados*, João Pessoa, Brasil, SBC, pp. 238–254.
2. Barsalou, L.; Breazeal, C.; Smith, L. (2007). "Cognition as coordinated non-cognition". *Cognitive Processing,* 8(2), pp. 79-91.
3. Beech, D. (1988). "A foundation for evolution from relational to object databases". In *Extending Database Technology*, Schmidt, J.W.; Ceri, S.; Missikoff, M. (eds.). New York, Springer, pp. 251–270.
4. Bobrow, D. G.; Winograd, T. (1977). "An overview of KRL-0, a knowledge representation language". *Cognitive Science* 1,1, pp. 3–46.
5. Booth, W. (1974). *A Rhetoric of Irony*. U. of Chicago Press.

6. Breitman, K.; Casanova, M.A.; Truszkowski, W. (2007). *Semantic Web: Concepts, Technologies and Applications*. London, Springer.
7. Burke, K. (1969). *A Grammar of Motives*. U. of California Press.
8. Chandler, D. (2007). *Semiotics: The Basics*. Routledge.
9. Chen, P.P. (1976). "The entity-relationship model: toward a unified view of data". *ACM Trans. on Database Systems*, 1:1, pp.9–36.
10. Chen, P.P. (2006). "Suggested Research Directions for a New Frontier – Active Conceptual Modeling". *ACM-L Workshop*, *ER 2006*, LNCS 4215, Springer, pp. 1–4.
11. Ciarlini, A.E.M.; Barbosa, S.D.J.; Casanova, M.A.; Furtado, A.L. (2009). "Event Relations in Plan-Based Plot Composition". *ACM Computers in Entertainment*. To appear.
12. Codd, E. F. (1972). "Relational completeness of data base sublanguages". In *Database Systems*, Rustin, R. (ed.), Prentice-Hall, pp.65–98.
13. Damme, C.V.; Heppe, M.; Siorpaes, K. (2007). "FolksOntology: An Integrated Approach for Turning Folksonomies into Ontologies". Proc. *ESWC Workshop - Bridging the Gap between Semantic Web and Web 2.0, SemNet 2007*, pp. 57–70.
14. Date, C.J. (2003). *An Introduction to Database Systems*. Addison-Wesley.
15. Fauconnier, G; Turner, M. (2002). *The Way We Think*. New York, Basic Books.
16. Fillmore, C. (1968). "The case for case". In *Universals in Linguist Theory*, Bach, E.; Harms, R.T. (eds.). New York, Holt, pp. 1–88.
17. Furtado, A.L.; Casanova, M.A.; Barbosa, S.D.J.; Breitman, K.K. (2008). "Analysis and Reuse of Plots using Similarity and Analogy". Proc. *27th International Conference on Conceptual Modeling* (*ER 2008*), LNCS 5231, pp. 355-368.
18. Furtado, A.L.; Kerschberg, L. (1977). "An algebra of quotient relations". Proc. *ACM SIGMOD International Conference on Management of Data*, pp.1–8.
19. Jaeschke, G.; Scheck, H.J. (1982). "Remarks on the algebra of non first normal form relations". Proc. *1st ACM SIGACT-SIGMOD symposium on principles of database systems*, pp. 124–138.
20. Karlsson, B.F.; Furtado, A.L.; Barbosa, S.D.J.; Casanova, M.A. (2009). PMA: "A Plot Manipulation Algebra to Support Digital Storytelling". Proc. *8th International Conference on Entertainment Computing*, to appear.
21. Lakoff, G. (1987). *Women, Fire, and Dangerous Things*. The University of Chicago Press.
22. Minsky, M. (1975). "A Framework for Representing Knowledge". In Winston, P.H. (ed.) *The Psychology of Computer Vision*. New York, McGraw-Hill, pp. 211–277.
23. Rich, E. (1983). "Users are individuals – individualizing user models". *International Journal on Man-Machine Studies*, 18, pp. 199-214.
24. Saussure, F. (1916). *Cours de Linguistique Générale*. Bally, C. et al. (eds.), Payot.
25. Schank, R.C.; Colby, K. (eds.) (1973). *Computer Models of Thought and Language*. W.H. Freeman.
26. Smith, J.M.; Smith, D.C.P. (1977). "Data abstraction: aggregation and generalization". *ACM Transactions on Database Systems*, 2:2, pp.105–133.
27. Stonebraker, M. (1986). "Inclusion of New Types in Relational Data Base Systems". Proc. *Second International Conference on Data Engineering*, pp. 262–269.
28. Stonebraker, M.; Madden, S.; Abadi, D. J.; Harizopoulos, S.; Hachen, N.; Helland, P. (2007). "The end of an architectural era". Proc. *VLDB'07*, pp. 1150-1160.
29. Ullman, J. D.; Widom, J. (2008). *A first Course on Database Systems*. Prentice-Hall.
30. Varvel, D.A.; Shapiro, L. (1989). "The Computational completeness of extended database query languages". *IEEE Transactions on Software Engineering*, 15.5, pp.632–638.
31. Winston, M.E.; Chaffin, R.; Herrmann, D. (1987). "A taxonomy of part-whole relations". *Cognitive Science*, 11, 4.