

MULTI-SPACE NUMBERED DATA BASE

Krassimir Markov

Abstract: *This paper introduces the concept of a Multi-Space Numbered Data Base (MSNDB), a data storage paradigm in which raw values are stored without semantic annotation, and meaning is assigned exclusively by external interpretation agents at query time. Drawing on the theoretical foundations of the Multi-Domain Information Model (MDIM) and the FOI Archive Manager (FOI ArM), we propose a formal model comprising four independent components: a semantically neutral data store, an n -dimensional descriptor space, a context vector, and a selector script. The model supports simultaneous, divergent interpretations of the same datum by different programs, a property we term polycontextual interpretation, and defines a principled response, propagating an UNDEFINED marker, when no descriptor matches the current context. We describe the formal specification of each component, the YAML-based representation of descriptors and selectors, and the architecture of the complete system. The proposed approach extends MDIM with an application-level semantics layer and is directly implementable using the existing FOI ArM infrastructure.*

Keywords: *context-free access method, semantic neutrality, multi-domain information model, descriptor space, polycontextual interpretation, UNDEFINED propagation*

ITHEA Keywords: *E.1 Data Structures - Distributed data structures; E.2 Data Storage Representations - Composite structures*

DOI: <https://doi.org/10.54521/ijita31-02-p01>

1. Introduction

Conventional database systems attach semantic meaning to data at the point of storage. Column names, data types, constraints, and foreign-key relationships all encode assumptions about what a value represents. While this approach simplifies querying within a single application domain, it creates rigidity: the same numerical value, for instance, the height of a spirit column in millimeters, cannot simultaneously represent a temperature in degrees Celsius to one program and Fahrenheit to another without schema duplication or application-level overrides.

The analogy of a room thermometer is instructive. The physical instrument stores a single datum, the height of a spirit column in millimeters, yet two scales, Celsius and Fahrenheit, are printed on opposite sides of the glass. Neither scale is privileged; both are equally valid interpretations of the same underlying physical reality. The thermometer itself carries no semantic commitment: it is the observer who supplies the interpretive frame.

This paper formalizes that intuition into the database architecture we term a Multi-Space Numbered Data Base (MSNDB). In a MSNDB, the storage layer retains only raw values. Semantic meaning is supplied by external descriptor scripts, selected dynamically by a selector script that evaluates a context vector describing the requesting agent and its operating conditions. Two programs may, at the same instant, read the same record and produce different, yet equally correct interpretations.

The theoretical substrate for our model is the Multi-Domain Information Model (MDIM) introduced by Markov (1984, 2004a, 2004b) and Markov, Todorov, and Nikolov (1990) which provides a rigorous framework for n-dimensional information spaces, coordinate-addressed elements, and polyindexation. We show that a MSNDB is naturally expressible as an MDIM aggregate, and that the descriptor space itself constitutes a second MDIM information space addressed by context coordinates.

The Multi-Domain Information Model (MDIM) framework has been applied in practice to a range of information management challenges, including the

organization of data warehouses and operational data stores (Markov, 2005), the achievement of data independence in multi-dimensional numbered information spaces (Markov, 2008), and the storage of large-scale graph and RDF datasets using the Collect/Report Paradigm (Markov & Ivanova, 2015). A dedicated manipulation and query language for numbered information spaces, NISL, has been defined to support direct coordinate-based access, natural language addressing, and the Collect/Report Paradigm over information bases of practically unlimited size (Markov, Velychko, Ivanova, & Karastanev, 2016). These applications demonstrate the practical scalability of MDIM-based systems and motivate the extension proposed here.

The remainder of the paper is organized as follows. Section 2 situates the work in relation to existing approaches. Section 3 develops the formal model. Section 4 describes the YAML representation of system components. Section 5 addresses the UNDEFINED condition and its propagation semantics. Section 6 discusses the recursive symmetry of the architecture. Section 7 gives a practical account of building and operating an MSNDB application, including an Object Pascal/Delphi realization over the existing FOI ArM32 access method. Section 8 characterizes MSNDB as a database management system, describes how end users, query authors, and administrators each interact with it, and applies Natural Language Addressing to give the descriptor and selector space human-readable, directly addressable names at scale. Section 9 discusses broader implications and limitations, and Section 10 concludes with directions for future work.

2. Related Work

The relational model of Codd (1970) remains the dominant paradigm for structured data storage. Its strength lies in the declarative specification of data semantics through schema definitions. However, as Markov (2004a) demonstrates, the relational model cannot efficiently represent information spaces of range greater than three, polyindexation and the hierarchical chaining of index structures have no natural relational counterpart.

Semantic web technologies (Berners-Lee et al., 2001) pursue the complementary goal of attaching rich, machine-readable meaning to data. RDF and OWL embed semantics within the data layer itself, the inverse of our proposal. Context-aware computing (Dey, 2001) addresses the dynamic selection of behavior based on situational parameters, but typically operates at the application layer rather than the storage layer.

Polymorphic data types in programming languages (Cardelli & Wegner, 1985) allow a single value to be processed by multiple type-specific routines, which is conceptually close to our descriptor mechanism. The key difference is that programming-language polymorphism is resolved at compile time or runtime within a single program, whereas MSNDB interpretation is resolved by independent programs consulting shared external descriptors.

The concept of data independence in multi-dimensional numbered information spaces was analyzed by Markov (2008), who distinguished physical, logical, and distribution independence in the FOI ArM framework. The Cell Oriented Programming (CellPro) technology (Markov, Ivanova, & Mitov, 1995) provides a logical-level mapping layer between physical coordinate-addressed storage and user-visible views, a role that is structurally analogous to the selector and descriptor layers in the present model. In CellPro, cells are encapsulated agents that may contain data or executable functions and are connected to specific coordinates in the information space; scripts define fixed or variable configurations of cells that are activated sequentially or in event-driven fashion (Markov, Ivanova, & Mitov, 1999). The key distinction is that CellPro scripts are defined and interpreted within the FOI System Builder (FOI SyB) system, whereas in MSNDB the corresponding artifacts (descriptors and selectors) are fully external, coordinate-addressed, and independently deployable without modifying either the data store or the executing program.

The physical organization underlying these higher-level technologies is described by Markov (2004b), who specifies the block-level structure of the FOI ArM32 access method: archives as fixed-block direct-access files, elements as variable-length byte sequences addressed through coordinate arrays, and a

five-level tree of space and domain indexes mapping coordinates to physical block offsets. Two features of this organization are directly relevant to MSNDB.

First, the coordinate array convention, in which the zeroth array position specifies the number of coordinates that follow, and leading high-order positions are addressed first, corresponds exactly to the space address $A = (c_n, \dots, c_1)$ used in Section 3.3 to define the context vector and descriptor coordinates.

Second, the explicit assumption that archive space is sparsely populated, with no storage cost for empty coordinates, is the same assumption underlying the sparse representation of DESC in Section 3.5.

The dialogue-level realization of this access method is described by Ivanova, Mitov, and Markov (2004), who show that a complete family of business applications, called Complex FOI, can be built from only four end-user interaction structures (menu, hierarchy, table, and text file) layered over the coordinate-addressed FOI ArM storage, without requiring any additional interface primitives.

It is worth distinguishing, at this point, which access method realization the examples in this paper actually depend on. The Complex FOI accounting system and the Object Pascal realization of Section 7.5 are existing, deployed artifacts and are described as they are: built against FOI ArM32, the 32-bit access method whose 4-gigabyte archive and 2^{32} -byte element limits (Section 2) were entirely adequate for coordinate vectors whose components are small integers, such as TypOp and Mon. MSNDB as such, however, and in particular the Natural Language Addressing technique of Section 8.5, is intended to run on FOI ArM64, popularly known as FOI BigArM: the 64-bit successor to ArM32, offering an address space and per-element capacity reported to be several orders of magnitude larger. The distinction matters concretely for letter-coded coordinates. A coordinate produced by encoding a short path such as /accounting/bg/vat/standard_rate one character per axis is already a coordinate array dozens of components long, each component itself a 32-bit (or, for extended character sets, wider) code; a 32-bit-oriented access method bounds both how long a name can be encoded and how many such long coordinates an archive can hold before exhausting its addressing capacity,

whereas FOI ArM64 was designed with this requirement in view and removes the bound for all but the most extreme path lengths. Throughout the remainder of this paper, FOI ArM32 should accordingly be read as the access method underlying the historical Complex FOI example and the Pascal listings of Section 7.5, while FOI ArM64/BigArM should be understood as the intended substrate for MSNDB deployments that make use of Natural Language Addressing as described in Section 8.5.

Examination of the actual FOI ArM32-based accounting system within Complex FOI reveals an important precursor to the MSNDB model described in this paper. In the current implementation, the programmer writes a structured commentary block, informally called a descriptor, directly in the source code, enumerating the meaning of each coordinate address in each archive. For example, the accounting transaction archive ACCxxTyy.DAT uses the following coordinate-to-meaning mapping (excerpt):

ACCxxTyy.DAT

(1, 1) - active month

(1, 18) - NOK class

(1, 19) - BULSTAT (company registration number)

(1, 20) - internal archive name, e.g. ACC26T01.DAT

(1, 24) - VAT registration number

(1, 25) - company name

(1, 26) - address

(1, 27) - bank account

(1, 91) - VAT standard rate (20%)

(1, 92) - VAT tourist rate (7%)

...

Regular entries:

$f_dom = 10 + 20 * TypOp + 60 * Mon;$

-- TypOp in {0,1,2}, Mon in {0..12}

Offset within f_dom:

0 - account

1 - type

2 - number

3 - date

4 - total amount

5 - VAT number

6 - transaction type

7 - tax

8 - classification number ...

The formula $f_dom = 10 + 20 * TypOp + 60 * Mon$ is particularly instructive. It constitutes an implicit selector: the context vector (TypOp, Mon), encoding the operation type (income/expense/other) and the accounting month, is mapped by arithmetic to a domain coordinate within the archive space. Each value of f_dom addresses a distinct zone of the archive that holds transaction records of a specific type for a specific month. This is precisely the function $S: K \rightarrow \omega$ described in Section 3.6, except that in the current implementation it exists only as a formula in a source-code, not as an external, independently deployable script.

Similarly, the commentary block listing the meaning of each (space, coordinate) pair functions as an informal descriptor: it records the transformation from raw byte sequences at coordinate positions to semantically meaningful fields such as company name, VAT rate, or bank account number. The descriptor is embedded in the source code and is therefore inseparable from the program that reads it; it cannot be updated without recompiling the application, cannot be shared across programs, and does not support the UNDEFINED response, if a

coordinate carries no descriptor entry, the programmer must handle the case ad hoc.

The MSNDB model proposed in this paper is therefore a direct formalization and generalization of this existing practice. The hand-written commentary becomes an external YAML descriptor stored in DESC at the coordinate vector corresponding to the relevant context. The arithmetic formula becomes an external selector script S. The program that previously contained all three layers (storage access, descriptor, and selector) is reduced to the universal five-step interpreter of Section 3.7. The accounting system of Complex FOI thus serves not only as an existence proof that coordinate-addressed polycontextual storage is practically viable at scale, but also as the direct motivating instance for the architectural separation that MSNDB formalizes.

To our knowledge, the explicit separation of raw storage from dynamically selected, externally stored, coordinate-addressed semantic descriptors, and the treatment of the descriptor repository itself as a context-free n -dimensional information space, has not previously been proposed in the literature.

3. Formal Model

3.1 Foundational Concepts from MDIM

We briefly recall the definitions from Markov (2004a) that underpin our model. A basic information element e is an arbitrary string of indivisible information fragments. One-dimensional information space is a triple $S_1 = (E_1, \mu_1, C_1)$, where E_1 is a set of elements, C_1 is a subset of $\mathbb{N}$ (natural numbers) and serves as a set of coordinates, and $\mu_1: E_1 \leftrightarrow C_1$ is a bijection. An information space of range $n \geq 2$ is defined recursively: $S_n = (E_n, \mu_n, C_n)$, where each element of E_n is itself an information space of range $n-1$. A space address of range n is a sequence $A = (c_n, c_{n-1}, \dots, c_1)$ with $c_i \in C_i$. An aggregate $D = (G, \tau)$ is a set G of information spaces together with a set τ of mappings from one designated main space into the others.

3.2 The Data Store

The data store DB is an information space of range 1:

$$DB = \{ (i, v) \mid i \in N, v \in V \}$$

where V is a set of raw values (for instance, integers, reals, or byte strings). DB carries no column names, no type annotations, and no space-specific constraints. It records only the bijective association between a coordinate i and a raw value v . This is the minimal information space in the sense of MDIM: semantically inert storage.

3.3 The Context Vector

A context K is a tuple of coordinate values over a fixed set of named axes:

$$K = (k_n, k_{n-1}, \dots, k_1) \in N^n$$

Each axis represents one dimension of the operating environment of the requesting agent, for example, application space, geographic region, user role, or operational mode. K is itself a point in an n -dimensional integer lattice $\Omega = N^n$. Crucially, K encodes no information about the data store and no information about available descriptors; it describes only the agent and its situation.

The position of each coordinate within the vector carries structural meaning. Given $A = (c_n, c_{n-1}, \dots, c_2, c_1)$, coordinate c_i at position i counted from the end of the vector addresses an information space of rank i in the sense of MDIM. Thus c_n addresses the highest-rank space (rank n), and c_1 addresses the lowest-rank space (rank 1 is a basic information domain). This alignment with the MDIM definition of space address of range n (Markov, 2004a) means that the context vector is not merely an identifier but a structural path through the hierarchy of information spaces.

3.4 The Descriptor

A descriptor D is a record stored at a point $\omega \in \Omega$ in the descriptor space. Formally:

$$D = (\omega, f, f_{inv}, input_{space}, range, label, unit)$$

where ω is the coordinate vector identifying this descriptor, $f: V \rightarrow R$ is the forward transformation from raw value to interpreted value, $f_{\text{inv}}: R \rightarrow V$ is the inverse transformation (enabling write-back), $\text{input}_{\text{space}}$ is a subset of V containing the valid raw inputs, range is a subset of R containing the valid interpreted outputs, label is a human-readable description, and unit is the name of the interpretive unit (e.g., degC, cm, USD). The existence of f_{inv} ensures that the system can accept an interpreted value from a user and convert it to raw storage form without loss.

3.5 The Descriptor Space

The set of all descriptors forms an information space of range n in the sense of MDIM:

$$\text{DESC} = \{ (\omega, D) \mid \omega \in \Omega \}$$

DESC is a sparse structure: most points in Ω are empty. This is consistent with MDIM's assumption that all information elements exist in principle but most spaces are empty in practice (Markov, 2004a). Physical storage uses a special sparse representation, so that only populated points consume resources. Because DESC is itself an information space addressed by coordinate vectors, it can be managed by the same access method used for DB, achieving architectural uniformity.

3.6 The Selector

The selector S is a script, an executable artifact external to both DB and DESC, that maps a context vector to a descriptor coordinate:

$$S: \Omega \rightarrow \Omega \cup \{ \text{UNDEFINED} \}$$

Internally, S is an ordered sequence of condition-result rules. Each rule tests whether K satisfies a predicate P_i over the axis values; the first matching rule returns the corresponding descriptor coordinate ω_i . If no rule matches, S returns UNDEFINED. The selector has no knowledge of DB and no knowledge of the raw values it will be applied to; it operates purely on the context.

3.7 Read and Write Operations

The complete system constitutes an MDIM aggregate:

$$\text{AGG} = (\{ \text{DB}, \text{DESC}, \text{CONTEXT} \}, \tau)$$

where τ consists of the mappings realized by the selector and the descriptor transformations. The fundamental operations are:

WRITE(i, r, K):

$\omega := S(K)$

if $\omega = \text{UNDEFINED}$:

return $\text{UNDEFINED}(K)$

$D := \text{DESC}(\omega)$

$\text{DB}(i) := D.f_{\text{inv}}(r)$

READ(i, K):

$\omega := S(K)$

if $\omega = \text{UNDEFINED}$:

return $\text{UNDEFINED}(K)$

$D := \text{DESC}(\omega)$

$v := \text{DB}(i)$

return $\text{DEFINED} (D.f(v), D.\text{label}, D.\text{unit})$

The result type of READ is therefore:

$\text{Result} ::= \text{DEFINED} (\text{value}, \text{label}, \text{unit}) \mid \text{UNDEFINED} (\text{context_vector})$

3.8 Relation to the Numbered Information Spaces Language

The formal READ and WRITE operations defined above are not merely abstract; they correspond directly to primitive operations already defined in the Numbered Information Spaces Language (NISL) (Markov, Velychko, Ivanova, & Karastanev, 2016). NISL is organized into three sub-languages: a definition language (NISDL), a manipulation language (NISML), and a query language (NISQL), operating over basic information elements addressed by coordinate arrays, with indexes and meta-indexes as auxiliary structures. The coordinate array used by NISL to address an element, with the space range given as its first component, is structurally identical to the space address $A = (c_n, c_{n-1}, \dots, c_1)$ introduced in Section 3.3.

This correspondence suggests a direct implementation strategy: DB and DESC, both being numbered information spaces, can be realized as two NISL-addressable spaces within a single information base. The READ operation becomes a NISML read on DB at coordinate i , combined with a NISML read on DESC at coordinate ω returned by the selector; the WRITE operation becomes the corresponding pair of writes. Because NISL operations are explicitly described as context-dependent, requiring specialized realizations for concrete purposes such as search, sorting, or reporting, the selector and descriptor mechanisms of MSNDB can be regarded as a particular instance of such a context-dependent NISL realization, specialized for semantic interpretation rather than search or reporting.

The three sub-languages address distinct phases of the MSNDB lifecycle, and each maps naturally onto one of the operations defined in Sections 3.2–3.7.

3.8.1. NISDL (definition)

The definition sub-language declares the shape of a numbered information space before any element is stored in it: its range n , the cardinality (or open-endedness) of each coordinate axis, and the byte-level type of the elements it will hold. In MSNDB terms, an NISDL declaration brings a space into existence as an addressable target for DB or DESC. A schematic declaration for the descriptor space DESC introduced in Section 3.5 takes the form

DEFINE SPACE DESC (RANGE = n, ELEMENT = VARBYTE);

which reserves an n-dimensional, practically unbounded, sparsely populated coordinate space whose elements are variable-length byte sequences – exactly the structure required to hold serialized descriptor records D at coordinates ω .

3.8.2. NISML (manipulation)

The manipulation sub-language performs the element-level operations enumerated for ArM32 in Section 2 – read, write, append, insert, cut, replace, and delete – against a coordinate array. Both primitive operations of Section 3.7 reduce directly to NISML statements. The READ(i, K) operation, once the selector has resolved K to a coordinate ω , becomes a pair of NISML reads:

READ ELEMENT FROM DESC AT (n, ω_n , ..., ω_1)

INTO :descriptor_buffer;

READ ELEMENT FROM DB AT (1, i) INTO :raw_buffer;

followed by application of the transform f carried in :descriptor_buffer to :raw_buffer. The WRITE(i, r, K) operation is the symmetric pair of NISML writes, applying f^{-1} before the WRITE ELEMENT INTO DB statement. Because NISML addresses elements purely by coordinate array, it is agnostic to whether the array designates a position in DB or in DESC; the same primitive serves both, which is the operational basis for the architectural symmetry developed in Section 6.

3.8.3. NISQL (query)

The query sub-language operates at the level of subspaces rather than single elements, supporting the projection and existence operations introduced for ArM32 indexes in Section 2 (next/previous present, next/previous empty, count, delete subspace) together with the natural-language addressing facility of the Collect/Report Paradigm (Markov, Ivanova, Velychko, Vanhoof, & Castellanos, 2015; Markov & Ivanova, 2015). Two NISQL capabilities are used directly by MSNDB.

First, partial-coordinate projection answers the administrative queries mentioned in Section 6, for example:

```
SELECT ELEMENT FROM DESC WHERE COORD(1) = "medicine";
```

which retrieves every descriptor whose first axis is fixed to the medicine space, regardless of the values of the remaining axes – precisely the projection $[1, *, *, *]$ discussed informally in Section 6.

Second, the existence test `EXISTS ELEMENT AT (...)` is the NISQL primitive that the selector implementation invokes immediately before returning a resolved coordinate, so that an absent descriptor is detected and reported as `UNDEFINED(K)` rather than triggering a runtime error when the subsequent NISML read is attempted. In this sense, NISQL supplies the verification step that makes the `UNDEFINED` semantics of Section 5 enforceable at the language level rather than merely at the level of the formal model.

4. YAML Representation of System Components

4.1 Rationale for YAML

YAML (YAML Ain't Markup Language) was chosen for the external representation of descriptors and selectors because it is human-readable. It supports hierarchical structure without the verbosity of XML, and is natively parsed by all major programming environments. The choice reflects the design principle that descriptors and selectors should be authored and audited by domain experts, not only by software engineers.

4.2 Descriptor Schema

A descriptor file encodes a single point in DESC. The coordinate vector appears as the `id` field, making the filename and the logical address identical:

descriptor:
id: [1, 1, 1, 1]
version: "1.0"
label: "Child height (pediatrics, BG)"
unit: "cm"

input:
type: float
range: [0.0, 100.0]

output:
type: float
range: [0.0, 200.0]

transform:
expression: " $v * 0.435$ "

inverse:
expression: " $v / 0.435$ "

undefined:
policy: propagate

4.3 Selector Schema

The selector script defines the axis vocabulary and the ordered rule set mapping context tuples to descriptor coordinates:

selector:
version: "1.0"

axes:
- axis: 1
name: "space"

```
      values: {1: "medicine", 2: "meteorology", 3: "finance"}
- axis: 2
      name: "region"
      values: {1: "BG", 2: "US", 3: "EU"}
- axis: 3
      name: "user_role"
      values: {1: "physician", 2: "engineer"}
- axis: 4
      name: "mode"
      values: {1: "paediatrics", 2: "cardiology"}
rules:
  - context: {space: "medicine", region: "BG",
user_role: "physician", mode: "paediatrics"}
  descriptor: [1, 1, 1, 1]
  - context: {space: "meteorology", region: "US"}
  descriptor: [2, 2, 0, 0]
  fallback: UNDEFINED
```

Axes with value 0 in a descriptor coordinate act as wildcards, indicating that the corresponding dimension is not discriminative for that descriptor.

4.4 Case Example: Formalizing the Complex FOI Accounting Archive

To illustrate the transition from the current hand-written descriptor practice to the MSNDB model, we show how the accounting transaction archive of Complex FOI (Section 2) would be expressed in the YAML descriptor schema. The context vector has two axes: operation type (TypOp in {0,1,2}) and month (Mon in {0..12}). The selector maps these to the archive domain coordinate via the formula $f_{\text{dom}} = 10 + 20 * \text{TypOp} + 60 * \text{Mon}$. In MSNDB terms, this becomes:

selector:

version: "1.0"

axes:

- axis: 1

name: "TypOp"

values: {0: "income", 1: "expense", 2: "other operations"}

- axis: 2

name: "Mon"

values: {0: "annual", 1: "Jan", 2: "Feb", ..., 12: "Dec"}

rules:

- context: {TypOp: "*", Mon: "*"}

descriptor_formula: "10 + 20 * TypOp + 60 * Mon"

fallback: UNDEFINED

The static fields (company name, VAT number, bank account, etc.) are represented as fixed-coordinate descriptors:

descriptor:

id: [1, 19]

label: "Company BULSTAT registration number"

unit: "string"

transform:

expression: "raw_string(v)"

inverse:

expression: "raw_string(v)"

descriptor:

id: [1, 24]

label: "VAT registration number"

unit: "string"

transform:

expression: "raw_string(v)"

descriptor:

id: [1, 91]

label: "VAT standard rate"

unit: "%"

transform:

expression: "v / 100.0"

inverse:

expression: "v * 100.0"

input:

domain: [0.0, 100.0]

The context-dependent transaction records, whose domain coordinate is computed by f_{dom} , are described by a parametric descriptor template:

descriptor_template:

id_formula: "10 + 20 * TypOp + 60 * Mon"

axes: [TypOp, Mon]

offsets:

- | | | |
|----|----------------------------------|------------------|
| 0: | {label: "account", | unit: "string"} |
| 1: | {label: "transaction type", | unit: "code"} |
| 2: | {label: "document number", | unit: "integer"} |
| 3: | {label: "date", | unit: "date"} |
| 4: | {label: "total amount", | unit: "BGN"} |
| 5: | {label: "VAT registration no.", | unit: "string"} |
| 6: | {label: "deal type", | unit: "code"} |
| 7: | {label: "tax amount", | unit: "BGN"} |
| 8: | {label: "classification number", | unit: "integer"} |

The `descriptor_template` above deserves one further clarification, because it is easy to misread it as describing a single table with twenty columns in the relational sense, which is not quite what the underlying archive does. Each value of `f_dom` does not address a two-dimensional table; it addresses the first of twenty separate one-dimensional spaces, one per offset, laid out contiguously: offset 0 (account) occupies the one-dimensional space at coordinate `f_dom`, offset 1 (transaction type) occupies the space at `f_dom + 1`, and so on through offset 19. A transaction record, in the sense of one line of a memorial order, is not a physically contiguous row at all; it is the set of elements that happen to share the same second coordinate, the record number, across all twenty of these one-dimensional spaces. Reading record number `r` therefore means reading the same coordinate `r` independently from each of the twenty spaces `f_dom`, `f_dom + 1`, ..., `f_dom + 19`, and assembling the results; writing an update to a single field of that record means writing to exactly one of those twenty spaces, at coordinate `r`, without touching the other nineteen.

This is the precise sense in which `f_dom` selects a *zone* rather than a table: the formula `10 + 20*TypOp + 60*Mon` picks out the first coordinate of a block of twenty contiguous one-dimensional spaces reserved for one (operation type, month) combination, and the `descriptor_template`'s `offsets` field enumerates which of those twenty spaces, relative to `f_dom`, holds which field. The practical consequence, developed further in Section 7.5 and Section 9, is that a program never needs to read or write an entire record to access or update one field of it: it computes `f_dom + offset` for the field of interest, and performs a single-coordinate read or write directly at `(f_dom + offset, r)`. This is a direct field-level access discipline rather than a row-level one, and it is the architectural source of the update-cost advantage discussed in Section 9.

This YAML representation makes explicit what was previously implicit in source-code comments and arithmetic formulas: the context vector, the selector mapping, and the semantic meaning of each coordinate offset are now independent, versioned artifacts that can be shared across programs, updated without recompilation, and validated against the UNDEFINED propagation rules of Section 5.

5. The UNDEFINED Condition and Propagation Semantics

5.1 Motivation

A naive implementation might respond to a missing descriptor by substituting the nearest available descriptor (nearest-neighbor fallback) or by raising a fatal error. Neither behavior is satisfactory. Nearest-neighbor substitution introduces silent semantic errors that are difficult to detect and may propagate through downstream computations. Fatal errors halt processing unnecessarily when the missing interpretation affects only a subset of the data being processed.

We adopt a third approach, motivated by the behavior of partial functions in mathematics and by the concept of nullable types in programming language theory: the system returns an explicit UNDEFINED marker carrying the offending context vector, continues processing the remaining data, and reports the complete set of unresolved contexts to the caller at the end of the computation.

5.2 Formal Propagation Rules

Let a computation C process a sequence of values $v_1, v_2, \dots, v_m$ under context K . Define an accumulator $U := \emptyset$. For each v_i :

$r_i = \text{READ}(i, K)$

if $r_i = \text{DEFINED}(*)$: use r_i normally

if $r_i = \text{UNDEFINED}(K)$: add K to U ; mark position i as undefined; continue

At the conclusion of C :

if $U = \emptyset$: result is complete

if $U \neq \emptyset$: result is partial;

report U to caller

The caller, a program or an end user, receives not merely a notification that data is missing, but the precise set of context vectors for which descriptors must be created. This transforms an opaque error into an actionable diagnostic.

5.3 Handling by the Program

In practice, programs operate in a three-tier response hierarchy. First, the program attempts to resolve the missing context using an internally available alternative descriptor (a local fallback). If no local fallback exists, it marks the affected positions as UNDEFINED and continues. Only when the computation is complete and $U \neq \emptyset$ does the program surface the unresolved contexts to the user. This hierarchy ensures that the user interface is not flooded with per-value notifications while preserving complete auditability of interpretive gaps.

6. Architectural Symmetry and Recursive Structure

A notable property of the MSNDB architecture is its recursive symmetry: the descriptor repository DESC is itself a context-free information space. It stores descriptors - structured values - addressed by coordinate vectors, without any intrinsic knowledge of the programs that will use them or the data they will be applied to. The same access method that retrieves raw values from DB retrieves descriptors from DESC. The same UNDEFINED propagation semantics that apply to data queries apply to descriptor lookups.

This symmetry has practical consequences.

First, DESC can scale to millions of descriptors without requiring a file-per-descriptor organization: the sparse n-dimensional structure handles the coordinate space efficiently, as demonstrated in MDIM implementations including the FOI BigArM (FOI ArM64) realization, whose address space is reported to exceed that of FOI ArM32 by several orders of magnitude and which underlies large-scale RDF dataset storage (Markov & Ivanova, 2015).

Second, DESC can itself be queried by partial coordinate vectors (projections), enabling administrative operations such as

'retrieve all descriptors for space = medicine'

without full-coordinate specification. Third, the selector S, being an external script, can itself be versioned, replaced, or composed with other selectors without touching either DB or DESC.

The resulting architecture has five independent components -- DB, DESC, K, S, and the executing program P, with the invariant that no component has compile-time knowledge of any other's content. The program P reduces to a five-step universal interpreter: read context, invoke selector, retrieve descriptor, read raw value, apply transformation. This universality means that a single program implementation can serve arbitrarily many interpretive spaces simply by deploying new descriptor and selector scripts.

This universality echoes the design philosophy of CellPro (Markov, Ivanova, & Mitov, 1995; 1999), in which the FOI SyB tool serves as a universal interpreter of cell scripts: the same runtime executes arbitrarily different information management applications by loading different script configurations. The MSNDB model extends this philosophy one level further by externalizing not only the application logic (the selector) but also the semantic mapping layer (the descriptor), making both independently versioned, deployable, and auditable artifacts.

7. Working with MSNDB

The preceding sections define MSNDB as a formal model and show its correspondence to existing MDIM and NISL infrastructure. This section gives a practical, end-to-end account of how an information system is built and operated on top of an MSNDB, organized around the four activities a development team actually performs: provisioning the spaces, authoring descriptors, authoring selectors, and operating the running system.

7.1 Provisioning

Setting up a new MSNDB application begins with two NISDL declarations: one for the raw data store DB and one for the descriptor space DESC, each with a range appropriate to the application's addressing needs. A typical accounting application following the Complex FOI pattern of Section 2 might declare

```
DEFINE SPACE DB ( RANGE = 2, ELEMENT = VARBYTE );
```

```
DEFINE SPACE DESC ( RANGE = 2, ELEMENT = VARBYTE );
```

No further schema commitment is made at this stage: neither space carries column names, value types beyond the byte-level element type, or constraints. Provisioning an MSNDB application is therefore a one-time, schema-light step, in marked contrast to relational system design where the bulk of up-front effort is schema definition.

7.2 Authoring descriptors

Descriptor authoring is the step that replaces the hand-written commentary practice illustrated in Section 2. A domain expert – an accountant, a clinician, a meteorologist – identifies the coordinates of interest and writes one YAML descriptor file per coordinate or per coordinate family, following the schema of Section 4.2. For a fixed coordinate, this is the single-descriptor form shown in Section 4.2 and 4.4. For a coordinate family parameterized by a formula, such as the accounting transaction zones addressed by `f_dom` in Section 4.4, the `descriptor_template` form is used instead, so that one authored artifact covers every (TypOp, Mon) combination without per-month duplication. Descriptors are stored as elements of DESC at the coordinate vector they describe, using the NISML write operation of Section 3.8; from that point onward they are available to every program with read access to DESC, independently of which program created them.

7.3 Authoring selectors

A selector is authored once per distinct way of resolving context to descriptor coordinates – typically once per application, though large organizations may compose several selectors (for example, a tenant-level selector chained to a department-level selector). The selector schema of Section 4.3 covers the enumerated-rule case; the formula-based case of Section 4.4 covers contexts whose resolution is naturally arithmetic rather than enumerable, such as a continuous range of months or operation codes. Both forms terminate in an explicit fallback: UNDEFINED, which is mandatory under the model of Section

3.6. Selector authoring is the step at which an organization encodes its own resolution policy – which descriptor a Bulgarian physician sees versus a German engineer – without touching either the data store or the programs that will eventually call the selector.

7.4 Operating the system

At run time, every program interacting with the MSNDB follows the same universal five-step interpreter identified in Section 6: obtain the current context K (from the operating environment, the logged-in user, or an explicit parameter); invoke the selector to resolve K to a coordinate ω or to UNDEFINED; on success, retrieve the descriptor D at ω via NISML; read or write the raw value at the data coordinate i via NISML; and apply $D.f$ or $D.f^{-1}$. On the UNDEFINED branch, the program follows the three-tier response hierarchy of Section 5.3: attempt a local fallback descriptor if one is available; otherwise mark the position as undefined and continue; and only surface the accumulated set U to the end user once the overall computation completes.

Two operational consequences follow from this division of labor.

First, extending the system to a new context (a new region, a new department, a new unit convention) requires authoring a new descriptor and, if necessary, a new selector rule; it never requires modifying, recompiling, or redeploying the program itself.

Second, because DESC is addressed and queried by the same NISML/NISQL primitives as DB (Section 3.8), the administrative operations available for monitoring data growth (counting elements in a subspace, enumerating populated coordinates, projecting by partial address) are available without modification for monitoring the growth and coverage of the descriptor space itself, which in a mature deployment is the artifact administrators most need to audit.

7.5 Object Pascal / Delphi realization

Existing MSNDB-style applications, including the Complex FOI accounting system examined in Section 2, are written in Object Pascal against the

FOI ArM32 access method, where each archive is an instance of the class TArm and every coordinate-level operation takes a coordinate array of type PCoordArray (a pointer to an array of Cardinal, with the zeroth element giving the array's own length, as specified in the ArM32 reference). This subsection shows how the five-step interpreter of Section 8.4 is realized directly on top of this existing, unmodified API, so that introducing MSNDB requires no change to the access method itself – only to the layer immediately above it. As noted in Section 2, the listings below illustrate the principle against FOI ArM32, the access method this historical example actually uses; a deployment additionally using the Natural Language Addressing technique of Section 8.5 would run the identical procedures against FOI ArM64 (FOI BigArM) instead, with no change to the Pascal code other than the underlying TArm implementation it links against.

Two TArm objects are opened once, at program start-up, to represent DB and DESC as two independent archives:

```
var
    DB, DESC : TArm;
begin
    DB := TArm.Create('DATA.ARM');
    DESC := TArm.Create('DESC.ARM');
end;
```

Both archives are used purely through the standard coordinate-level operations of Section 2 (ArmRead, ArmWrite, and so on); TArm itself carries no knowledge of which archive is "raw data" and which is "descriptors" – that distinction exists only in the calling code, which is precisely the architectural separation MSNDB requires of the underlying access method.

The selector is a small, independently compiled or interpreted function that maps a context record onto a coordinate array. For the Complex FOI example of Section 4.4, where the selector is the formula

$$f_dom = 10 + 20 * TypOp + 60 * Mon,$$

the Object Pascal realisation is a direct transcription:

```
function SelectDomain(TypOp, Mon: Cardinal): Cardinal;
begin
    if (TypOp > 2) or (Mon > 12) then
        Result := UNDEFINED_DOMAIN
        // sentinel value, never a valid f_dom
    else
        Result := 10 + 20 * TypOp + 60 * Mon;
    end;
```

The only departure from the present hand-written practice of Section 2 is the explicit UNDEFINED_DOMAIN sentinel: in the current Complex FOI code the formula is applied unconditionally, so an out-of-range TypOp or Mon silently produces a wrong-but-valid-looking domain coordinate, whereas the MSNDB-disciplined version makes the absence of a defined mapping an explicit, checkable value, in line with Section 5.

The five-step interpreter of Section 8.4 is then a short procedure built entirely from the ARM32 primitives of Section 2, with no new access-level functionality:

```
function ReadInterpreted(RecNo: Cardinal; Offset: Cardinal; TypOp, Mon:
Cardinal;
    out Value: Double; out IsDefined: Boolean): Cardinal;
var
    f_dom : Cardinal;
        // first of the 20 one-dimensional spaces for this (TypOp, Mon)
    omega : Cardinal;
    pcaDesc : PCoordArray;
        // (2, 1, omega) -- range-2 coordinate into DESC
    pcaData : PCoordArray;
        // (2, f_dom+Offset, RecNo) -- range-2 coordinate into DB
    descBuf : TDescriptorRecord;
        // parsed YAML-equivalent descriptor record
    rawBuf : array[0..7] of Byte;
    rawLen : Cardinal;
begin
    IsDefined := False;
        // Step 1: context K = (TypOp, Mon) is already in the parameter list
```

```

        // Step 2: invoke the selector
f_dom := SelectDomain(TypOp, Mon);
if f_dom = UNDEFINED_DOMAIN then
begin
    Result := ARM_UNDEFINED; Exit;
    // UNDEFINED(K) of Section 3.7
end;
// Step 3: retrieve the descriptor D at omega from DESC
omega := f_dom + Offset;
    // the one-dimensional space for this field
pcaDesc := MakeCoordArray([omega, 1]);
    // (range=2, k2=omega, k1=1)
if DESC.ArmLength(pcaDesc) = 0 then
begin
    Result := ARM_UNDEFINED; Exit;
    // no descriptor at this coordinate
end;
DESC.ArmRead(pcaDesc, descBuf, SizeOf(descBuf), 0);
// Step 4: read the raw value v of this field, for record RecNo, from DB
pcaData := MakeCoordArray([omega, RecNo]);
    // (range=2, k2=omega, k1=RecNo)
rawLen := DB.ArmRead(pcaData, rawBuf, SizeOf(rawBuf), 0);
if rawLen = 0 then
begin
    Result := ARM_UNDEFINED; Exit;
    // field does not exist for this record
end;
// Step 5: apply the transform D.f carried in descBuf
Value := ApplyTransform(descBuf, rawBuf, rawLen);
IsDefined := True;
Result := ARM_OK;
end;
```

Four points of correspondence with the formal model are worth making explicit.

First, the coordinate array (ω , 1) passed to `DESC.ArmRead` and the array (ω , `RecNo`) passed to `DB.ArmRead` are both ordinary `PCoordArray` values of range 2, in exactly the convention specified for `ArM32` in Section 2: the zeroth array position carries the dimension, and the high-order coordinate (ω in both cases) is addressed first. No extension to `PCoordArray`, `ArmRead`, or any other `ArM32` primitive is required – `DESC` is simply a second archive holding descriptor records instead of business data, exactly as anticipated by the recursive symmetry argument of Section 6.

Second, the role played by ω itself differs from the simplified single-field reading of Sections 3 and 4: here, $\omega = f_dom + \text{Offset}$ is computed inside Step 3, not handed to the function directly, because what the caller selects via `TypOp` and `Mon` is not one element but a zone of twenty one-dimensional spaces (Section 4.4), and `Offset` then picks out which of the twenty holds the field of interest. `RecNo`, the second component of the coordinate array passed to `DB.ArmRead`, is the record number within that one-dimensional space, exactly the record number of one line of a memorial order. A caller wanting the full record therefore calls `ReadInterpreted` once per field, varying `Offset` from 0 to 19 while holding `RecNo`, `TypOp`, and `Mon` fixed, rather than performing a single multi-field read; this is what Section 9 calls field-level rather than row-level access.

Third, every exit through `Result := ARM_UNDEFINED` corresponds to a distinct cause of `UNDEFINED(K)` in Section 5.2: an out-of-range context (the selector itself fails), a context that resolves to a coordinate with no stored descriptor (`DESC.ArmLength` returns 0), and a data coordinate with no stored element (`DB.ArmRead` returns a zero length). A calling routine implementing the three-tier hierarchy of Section 5.3 distinguishes these only if it needs to; for most callers, it is sufficient to treat any `ARM_UNDEFINED` result as "add `RecNo`, `Offset`, `TypOp`, `Mon` to the accumulator `U` and continue", regardless of which of the three checks triggered it.

Fourth, for repeated access to the same data coordinate, for example, when a program scans an entire month of transactions, the coordinate-array form of

ArmRead can be replaced by the descriptor-based operations ArmSeek/ArmGet/ArmPut of Section 2 once the relevant zone of DB has been located once via ArmSeek, exactly as recommended in the ArM32 reference for repeated access to the same element. This optimization is transparent to the MSNDB layer: Steps 3–5 of the interpreter are unaffected by whether Step 4 is realized as a single ArmRead or as an ArmSeek followed by a sequence of ArmGet calls. The same is true in the other direction, for ArmWrite/ArmPut: updating one field of one record is a single write at (ω , RecNo), with no other coordinate touched, the architectural property that Section 9 contrasts with row-rewrite costs in relational storage.

8. MSNDB as a DBMS

Sections 3 through 7 define MSNDB as a formal model and an implementation pattern. This section steps back and asks a question that the formal treatment leaves implicit: in what sense is MSNDB a database management system at all, and what, precisely, plays the role that a relational engine, a query optimizer, or a transaction manager plays in a conventional DBMS? Answering this is necessary because MSNDB has no single component that a practitioner would point to and call "the DBMS", a fact that is a deliberate consequence of the architecture, not an omission.

8.1 Where the DBMS function resides

A conventional DBMS bundles four responsibilities behind one engine: physical storage management, schema enforcement, query processing, and concurrency control. MSNDB does not bundle these; it distributes them across the five components of Section 6, and "the DBMS" is the composite behavior that results when all five operate together, not any single one of them.

Physical storage management and concurrency control are supplied by the FOI ArM access method itself (Markov, 2004b): block allocation, the five-level index tree, and the request-queue synchronization protocol for single- and multi-process access are exactly the services a conventional DBMS storage engine provides, and MSNDB introduces no replacement for them.

Schema enforcement, by contrast, is deliberately not supplied by the storage layer at all; it is supplied by DESC, the descriptor space, which plays the role a system catalogue plays in a relational DBMS, except that it is itself an ordinary numbered information space rather than a privileged, fixed-schema metadata table.

Query processing is supplied jointly by the selector S, which performs the access-path selection a query optimizer performs (deciding which descriptor, and therefore which interpretation, applies), and by the NISQL sub-language of Section 3.8, which performs the subspace-level operations (projection, existence testing, enumeration) that a query language performs over a conventional schema.

The component that has no conventional counterpart is the context vector K. In a relational DBMS, the meaning of a query result is fixed once the schema and the query text are fixed; there is no run-time parameter that changes what a column means. In MSNDB, K is exactly such a parameter, and its presence is what makes the system polycontextual rather than merely schema-flexible. This is the precise sense in which Section 1's claim should be read: MSNDB is a DBMS whose catalogue (DESC) and whose access path selection (S) are both first-class, independently addressable, run-time-parameterized data, rather than compile-time or administrator-time fixed structures.

8.2 The end user as application user

For the great majority of end users, MSNDB is invisible, exactly as the engine underneath Complex FOI is invisible to the accountant who uses it. Section 2 already established that the entire Complex FOI product family is built from four dialogue structures (menu, hierarchy, table, text file) over the coordinate-addressed FOI ArM store (Ivanova, Mitov, & Markov, 2004); under an MSNDB realization, those same four structures remain the end user's complete interface. The difference invisible to the user but significant to the organization is what happens behind the table cell: where the present Complex FOI implementation hard-codes the meaning of, say, coordinate (1, 91) as "VAT standard rate" inside the program (Section 2), an MSNDB-based table cell

resolves that meaning at display time via the selector and the descriptor at the user's current context. A Bulgarian accounting clerk and an auditor working under a different jurisdiction's VAT convention can use the identical table screen and see correctly localized values, without the application having been written, or rewritten, to know about more than one jurisdiction.

This is the sense in which the end user "uses the DBMS": not by issuing queries, but by inhabiting a context. Logging in as a particular role, in a particular region, in a particular operating mode, is itself the act that supplies K; the rest of the five-step interpreter of Section 7.4 runs without further end-user involvement. When K resolves to UNDEFINED, the end user's experience is likewise mediated by the application rather than by raw error messages: following the three-tier hierarchy of Section 5.3, the application either silently substitutes a fallback or, only when no fallback exists and the computation is complete, presents the user with the precise list of unresolved contexts, for example, "no VAT descriptor is defined for region = RO", which is actionable information a relational NULL or a generic exception is not.

8.3 The end user as query author

A second class of end user, the analyst or power user who would write SQL or build a query-by-form interface against a relational system interacts with MSNDB directly through NISQL (Section 3.8). Because NISQL operates over subspaces rather than over named, schema-fixed tables, a query is always expressed relative to a context, explicitly or by inheriting the caller's current K. The projection example of Section 3.8,

SELECT ELEMENT FROM DESC WHERE COORD(1) = "medicine";

is the MSNDB analogue of browsing a relational catalogue for all tables in a given schema, except that the result is the set of interpretations available, not a set of tables. A query that retrieves interpreted data, rather than metadata, composes a NISQL subspace selection over DB with the same selector-then-descriptor resolution that the five-step interpreter performs internally; in effect, the query author is permitted to fix some axes of K explicitly (for example, region = "BG") while leaving others to be supplied by the ambient session

context. This is a deliberate departure from SQL practice, where the query and the interpretation of its result columns are decided once, at schema design time, by the database administrator rather than negotiated per query.

For users who prefer the relational mental model outright, a NISQL result set, being a coordinate-indexed collection of interpreted values produced by the five-step interpreter, is straightforward to materialize as an ordinary table: one row per record number, one column per requested field, with the descriptor's label and unit supplying the column header. This is not a new capability that MSNDB must invent; it is the same tabular export facility already present in Complex FOI, which, in addition to its native menu/hierarchy/table/text-file interface (Section 2), can render any report as a delimited or spreadsheet file for import into Excel or an equivalent platform, and which historically went further still by providing its own embedded spreadsheet view for browsing query results directly. Under MSNDB, the same export step amounts to iterating a NISQL result over (RecNo, Offset) pairs as in Section 7.5, writing each interpreted value into the corresponding spreadsheet cell; the interpretation work has already been done by the descriptor before the value ever reaches the export layer, so the exported table is meaningful to a spreadsheet user with no awareness of contexts, selectors, or coordinate spaces at all.

8.4 The programmer and administrator

For the developer and the system administrator, MSNDB is used in exactly the two capacities described in Section 7: authoring (provisioning spaces, writing descriptors and selectors) and operating (monitoring DESC's growth and coverage, diagnosing accumulated UNDEFINED sets, versioning selectors). Both capacities are themselves expressible as ordinary NISML/NISQL operations against DESC, which is the practical payoff of the architectural symmetry argued for in Section 6: there is no separate "DBMS administration tool" with its own data model to learn, because administering the descriptor space uses the identical primitives used to administer the data space.

8.5 Naming, curating, and governing the descriptor and selector space

The architectural minimalism that makes MSNDB powerful (anyone, at any time, may author a new descriptor or selector at any coordinate) is also its principal operational hazard at scale. A deployment with thousands of descriptors and dozens of selectors, authored over years by different teams, faces three distinct governance problems that the formal model of Sections 3–6 does not, by itself, solve.

The first is opacity of address. A bare coordinate vector such as [3, 7, 1, 12] carries no information about what it denotes; a programmer reading or writing such a literal in source code, or an administrator inspecting DESC directly, cannot tell from the coordinate alone whether it designates a VAT rate, a patient's blood pressure threshold, or a sensor calibration constant.

The second is collision risk. Because the coordinate space is shared and writable by any authorized agent, two teams working independently may choose the same coordinate for unrelated purposes, or, more insidiously, may choose adjacent coordinates for related purposes without realizing the relationship, defeating the locality property noted informally in Section 3.4 (semantically related descriptors clustering near one another in Ω).

The third is discoverability. An administrator who needs to audit "every descriptor concerning value-added tax across all regions" has no way to issue that request against raw coordinates without already knowing which coordinates are VAT-related – precisely the information the coordinate space does not encode.

All three problems are instances of a single missing layer: human-readable names for each coordinate that matters. The Natural Language Addressing (NLA) approach developed for the Multi-Domain Information Model (MDIM) (Ivanova, Vanhoof, Markov, & Velychko, 2013; Markov, Ivanova, Velychko, Vanhoof, & Castellanos, 2015) supplies exactly this, and it does so in a way that is unusually well suited to MSNDB: rather than building a separate, searchable catalogue that maps names to coordinates, NLA dispenses with search entirely by making the name itself the coordinate.

The principle, as originally stated for the underlying access method, is direct. A concept such as "Pirrin" is conventionally stored by maintaining an index couple ("Pirrin", address) and following it: first look up the name to find the address, and then read the address. NLA observes that if the name's letters are encoded as numbers (for instance, ASCII codes), those numbers are already a coordinate vector and can address the archive directly: "Pirrin", encoded as (80, 105, 114, 114, 105, 110), is read or written at exactly that coordinate, with no index, and no lookup, in between. For a human the address is shown as the name; for the machine it is the coordinate array obtained by encoding the name's letters. Because FOI ArM already permits an archive to hold more than one independent coordinate system over the same element space (Section 2), this is not a new access method: it is an additional coordinate system, alongside the numeric one used by ordinary descriptors, addressing the same archive DESC.

Applied to MSNDB, the path /accounting/bg/vat/standard_rate is encoded letter by letter into a coordinate array exactly as "Pirrin" is encoded above, and the descriptor itself, not a pointer to it, is stored directly at that coordinate. Retrieving the VAT descriptor by name is therefore not a query against a map; it is an ordinary NISML read at the coordinate the name encodes to, with the same constant-time cost as reading any other element of DESC (Section 3.8). The numeric coordinate produced by a selector formula, such as the `f_dom` of Section 4.4, and the letter-coded coordinate produced by encoding a path, are simply two different coordinate systems that may, by design, be made to resolve to the same point in DESC, or may be used independently, exactly as Section 2 permits.

This resolves the three governance problems without introducing a second source of truth. Opacity is resolved because every descriptor that matters to a human can be authored, from the start, under a name rather than a bare integer vector; tooling needs only to perform the same letter-to-number encoding the storage layer performs, which is a fixed, parameterless function, not a lookup that can fall out of date. Collision is structurally prevented for descriptors named under sufficiently specific paths, for the same reason two different English

words are not spelled identically: distinct names encode to distinct coordinates, so authoring a new descriptor under an already-used name simply means writing to the same point in DESC that the existing descriptor occupies, an overwrite that NISML can refuse or version exactly as it would for a numerically addressed coordinate. Discoverability over groups of related descriptors, such as "every VAT descriptor", is the one capability direct letter-coding does not give for free, since there is no coordinate that means "all paths beginning with /accounting/*/vat/"; where such prefix discovery is needed, it is supplied by the multi-layer organization NLA already provides for exactly this purpose (Markov, Ivanova, Velychko, Vanhoof, & Castellanos, 2015): a separate layer, itself NLA-addressed, whose elements list the children of a given path. Maintaining that listing is an explicit bookkeeping step at descriptor-authoring time, not a search performed at query time, and is required only for the discovery use case, not for ordinary name-based retrieval.

Selectors benefit from the identical treatment: a selector script for, say, the Bulgarian accounting tenant is stored directly at the coordinate encoding /accounting/bg/selector, and a program locates it by encoding that name, not by searching a registry, which gives the selector-chaining scenario anticipated in Section 7.3 (a tenant-level selector composed with a department-level selector) an explicit, name-addressed realization: the tenant selector's own script can simply encode the department selector's path to invoke it, in the same way any other NLA reference does.

Two consequences of this design are worth flagging as open questions rather than settled answers.

First, letter-coding spends one coordinate axis position per character of the name, so the rank of the resulting coordinate grows with name length; whether this is acceptable depends on the practical addressing capacity of the underlying access method, which is precisely why Natural Language Addressing under MSNDB is intended for FOI ArM64 (FOI BigArM) rather than the 32-bit FOI ArM32 used by the historical Complex FOI example of Section 2: BigArM was designed to accommodate long letter-coded coordinate arrays directly, whereas under ArM32 very long, deeply nested paths would need to be

hashed or abbreviated rather than encoded verbatim, trading the transparency of a literal name for a shorter coordinate.

Second, because two descriptors can collide only if their names are identical, naming discipline (consistent casing, agreed abbreviations, a controlled vocabulary for path segments) carries more of the collision-avoidance burden than it would under a separately indexed catalogue with explicit conflict detection at registration time; this is a curation responsibility for the organization, not a property the formal model can enforce, and is left for future specification alongside the discovery-layer bookkeeping mentioned above.

9. Discussion

Sections 6 through 8 moved progressively from formal property to engineering practice to systems-level characterization, and it is worth pausing to weigh what each actually bought us before turning to the broader implications below. The recursive symmetry argued for in Section 6 is, strictly, a property of the model rather than an empirical claim; its value is that it licenses the practical economies described in Section 7 (one access method, one set of NISML/NISQL primitives, no separate "metadata engine") rather than asserting anything about run-time performance. The Object Pascal realization of Section 7.5 is correspondingly modest in scope: it shows that the five-step interpreter imposes no new requirement on FOI ArM32, but it does not, by itself, demonstrate that resolving a context through a selector script is cheap relative to a direct, hard-coded field access of the kind Complex FOI uses today. Where the selector or the deserialization of a descriptor record sits on a hot path, the architecture trades a constant per-access overhead for the flexibility described throughout this paper, and which side of that trade an application should take is a deployment decision the model deliberately leaves open. Section 8's characterization of MSNDB as a DBMS is, similarly, a redescription of existing components in DBMS terms rather than a new mechanism; its purpose is to make explicit that nothing resembling a conventional catalogue or optimizer needs to be built, since DESC and S already play those roles, but it does not address concerns, such as query plan caching or cost-based selector composition, that a mature DBMS implementation would eventually need to

confront. With these qualifications in view, the remainder of this section considers the model's implications for several areas of information systems design.

The field-level access pattern described in Section 4.4 and realized in Section 7.5, in which a single coordinate ($f_dom + Offset$, $RecNo$) is read or written for one field of one record, has a consequence for update cost that is worth making explicit because it runs counter to a common intuition about coordinate-addressed storage being merely a more flexible but not more efficient alternative to relational tables. In a relational system, updating one field of one row typically cannot be done strictly in place: depending on the storage engine, the existing row version is marked obsolete and a new version is appended at the end of the table (or of its current page), after which every index on the table, not only on the updated column, must be revisited to point at the new location, and the obsolete versions must eventually be reclaimed by a separate vacuuming or garbage-collection pass. The cost of an update is therefore not local to the field being changed; it is proportional to the number of indexes on the table and to the bookkeeping needed to keep row identity stable for readers that have not yet seen the update. Under MSNDB, by contrast, a field has its own one-dimensional space, addressed directly by ($f_dom + Offset$) on one axis and $RecNo$ on the other; writing a new value for that field is an in-place `ArmPut` at that exact coordinate, exactly as Section 7.5 shows, with no other coordinate, and therefore no other field of the same record, touched, no index outside that one space to revisit, and no obsolete row version left behind to reclaim. This is not a claim that MSNDB outperforms a relational engine on every workload; sequential scans over many fields of many records, for instance, may favor a row-major physical layout that MSNDB's field-major organization does not provide. It is, specifically, an update-locality advantage that follows directly from treating each field as its own addressable space rather than as a column within a row that must be relocated as a unit.

The MSNDB model carries implications for several areas of information systems design. In multi-tenant and multi-regional systems, where the same data must be presented under different regulatory, linguistic, or domain conventions, the

model eliminates the need for schema variants or application-level branching. The context vector encodes the tenant or region; the selector routes to the appropriate descriptor; the program is uniform across all contexts.

In scientific data management, where the same measurement may be expressed in SI units, Imperial units, or domain-specific scales depending on the receiving laboratory, MSNDB provides a clean separation between the archival record (the raw instrument reading) and the reporting layer (the descriptor). Provenance is preserved: the raw value is never altered, and every derived value is traceable to its descriptor and context.

The model also has implications for data exchange. When two systems share a MSNDB, they need not agree on interpretation: each supplies its own context and receives values in its own preferred form. The shared artifact is the raw value and the coordinate-addressed descriptor space; the interpretive contract is negotiated per-session through the context vector.

The Collect/Report Paradigm (CRP) introduced by Markov and Ivanova (2015) offers a complementary perspective: in CRP, storage nodes are silent unless their content matches an incoming request, and data is addressed via natural-language paths rather than integer coordinates. Section 8.5 already adopts the underlying Natural Language Addressing technique for the descriptor and selector space itself, encoding names directly as coordinates rather than building a searchable index; what remains open is whether the same technique should also govern access to DB, the raw data store, where it would let an end user or query author address a business record by a meaningful name rather than a bare integer i . A MSNDB system could integrate more fully with CRP by treating each CRP node as a context-specific view over a shared raw data store, with the descriptor and selector layers providing the mapping between the raw coordinate space and the natural-language-addressed reporting layer; this fuller integration, beyond the descriptor-space application already given in Section 8.5, remains a direction for future work.

A limitation of the current model is that descriptors are assumed to be deterministic functions of the raw value. Stochastic interpretations, where the mapping from raw value to interpreted value is probabilistic, are not addressed.

Similarly, the model does not yet treat the case where the transformation requires access to values at other coordinates in DB (context-dependent transformations in the data sense, as opposed to the agent-context sense). These are directions for future formalization.

10. Conclusion

We have presented the Multi-Space Numbered Data Base (MSNDB), a data storage paradigm in which raw values are stored without semantic annotation and meaning is assigned dynamically by external descriptor scripts selected through a context-sensitive selector. The model is grounded in the Multi-Domain Information Model of Markov (1984, 2004a) and can be expressed as an MDIM aggregate.

Its principal contributions are:

- (1) The explicit separation of storage, description, and context into independent, interchangeable components;
- (2) The formalization of the descriptor as a coordinate-addressed element in an n-dimensional information space;
- (3) The definition of UNDEFINED propagation semantics that support partial computation without silent error; and
- (4) The observation that the descriptor repository is itself a context-free information space, yielding a recursively symmetric architecture.

Future work will address stochastic descriptors, cross-record transformations, descriptor inheritance along coordinate axes, and the formal specification of selector composition rules.

Acknowledgements

This work is financed by FOI COMMERCE Co.

MSNDB, MDIM, ArM, SyB, and COMPLEX FOI are trademarks and property of FOI COMMERCE Co. (Markov, Ivanova, Mitov, 1995, 1999)

All other products, registered trademarks and trademarks are property of their owners.

References

- Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The semantic web. *Scientific American*, 284(5), 34-43. <https://doi.org/10.1038/scientificamerican0501-34>
- Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), 471-522. <https://doi.org/10.1145/6041.6042>
- Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377-387. <https://doi.org/10.1145/362384.362685>
- Dey, A. K. (2001). Understanding and using context. *Personal and Ubiquitous Computing*, 5(1), 4-7. <https://doi.org/10.1007/s007790170019>
- Ivanova, K., Mitov, I., Markov, K. (2004). Organization of the user dialogue in Complex FOI [in Bulgarian]. *Proceedings of the Second International Conference "Information Research, Applications and Education" i.TECH 2004*, Varna, Bulgaria. FOI-COMMERCE, 150-155.
- Ivanova, K., Vanhoof, K., Markov, K., Velychko, V. (2013). Introduction to the Natural Language Addressing. *International Journal "Information Technologies & Knowledge"*, 7(2), 139-146.

- Markov K. (1984) A Multi-domain Access Method. Proceedings of the International Conference on Computer Based Scientific Research. PLOVDIV, pp. 558-563.
- Markov K. (2004a). Multi-domain information model. International Journal "Information Theories & Applications", 11(4), 303-308. <http://www.foibg.com/ijita/vol11/ijita11-4-p01.pdf>
- Markov K. (2004b). Coordinate-based physical organization for computer representation of information spaces [in Bulgarian]. Proceedings of the Second International Conference "Information Research, Applications and Education" i.TECH 2004, Varna, Bulgaria. FOI-COMMERCE, 163-172.
- Markov K. (2005). Building data warehouses using numbered information spaces. International Journal of Information Theories and Applications, 12(2), 193-199.
- Markov K. (2008). Data independence in the multi-dimensional numbered information spaces. International Journal of Information Technologies and Knowledge, 2(1), 87-94.
- Markov K., Ivanova K. (2015). General structure of Collect/Report Paradigm for storing and accessing Big Data. International Journal of Information Theories and Applications, 22(3), 266-276.
- Markov K., Ivanova K., Mitov I. (1999) Multiagent Information Service Based on Scripts. Proceedings of the VIII International Conference KDS, Ukraine, Krim, 1999.
- Markov K., Ivanova K., Mitov I.. (1995) Cell Oriented Programming. IJ ITA, 1995; 3(1), 3-8.
- Markov K., Ivanova K., Velychko V., Vanhoof K., & Castellanos J. (2015). Natural Language Addressing. ITHEA. ISBN: 978-954-16-0070-2.
- Markov K., Todorov T., Nikolov V. (1990) Multidomain Access Method for the IBM PC. Research in Informatics, Vol.3, Academie-Verlag Berlin, pp. 218-230.

Markov K., Velychko V., Ivanova K., Karastanev S. (2016) „NISL: The Numbered Information Spaces Language”, III International Scientific Conference on "Information technologies and interactions" (IT & I), 2016, Kyiv, Ukraine, pp. 145-146.

Authors' Information



Krassimir Markov – Prof. Dr., Institute of Information Theories and Applications; P.O. Box: 775, Sofia-1090, Bulgaria;

e-mail: markov@ithea.org, itheaiss@gmail.com,

Major Fields of Scientific Research: General theoretical information research, Multi-dimensional information systems