WHITE PAPER

The HGDM Protocol
(Hypergraph Data Modeling)

Unifying Hypergraph Database Architecture with Scientific Research-Oriented Data Sharing

Introduction

Recent years have seen a proliferation of research in bioinformatic systems or "biomedical knowledge engineering," yielding new insights into optimal database design and data-mining strategies.[1] What has been emphasized as a consequence this research is that clinically and scientifically important information is intrinsically *heterogeneous*, spanning a diversity of formats and subject areas (including bioimaging, genomics, epidemiology, biochemistry, sociodemographics, immunology, clinical outcomes, and patient "quality of life" or related patient-centered evaluations). Medical data sharing — in addition to the quotidion exchange of patient histories for "real time" patient care (which is governed by strict regulations that inhibit novel, experimental technologies) — encompasses use-cases such as sharing observational studies, case series, clinical trial results, evaluating patient outcomes and cost/benefit analysis for different treatment options, and similar kinds of clinical, interventional, diagnostic, epidemiological, or translational research. In this research-oriented setting, there are still guidelines and requirements in effect (e.g., patient data must be suitably anonymized, and the reporting for clinical trials must accurately reflect trial design); but there is nonetheless considerable flexibility in how research-oriented medical data can be structured, modeled, and communicated.

In order to integrate all available information relevant for a given biomedical research/scientific project — both within single institutions and across multiple institutions — technology must be flexible enough to adapt to different data formats and profiles. This has led to two related (but philosophically distinct) paradigms: first, large-scale adoption of "Semantic Web" techniques oriented to knowledge-acquisition and Artificial Intelligence; and, second, the emergence of "hypergraph" database engines, which aspire to unify many different database architectures into a multi-purpose totality.

Both of these approaches offer the promise of a more tightly integrated biomedical information ecosystem: data and files reflecting many disparate scientific paradigms accessible through one platform that is shared across multiple institutions. With robust multi-institutional data sharing in place, technology can start to redress shortcomings in existing biomedical research practices. To wit, more extensive documentation of patient outcomes can assess the effectiveness of treatments within a broad spectrum of quality-of-life concerns (not just noting short-term clinical consequences of treatment modalities); and the merging of observational studies may provide a statistically diversified array of interventions and demographic/diagnostic profiles to compare/contrast, which can compensate for a paucity of clinical trials in emergent subject areas (Covid-19 being a case in point).

Despite these potential benefits, the large majority of biomedical data continues to be stored via conventional database and/or filesystem technologies. Moreover, there are nontrivial differences between how each hypergraph database engine is designed, as well as between hy-

[1] See e.g. (among many works that could be cited) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3555311/ or http://nemo.nic.uoregon.edu/wiki/images/8/88/978-3-938793-98-5_Munn_Ontology.pdf.

pergraph database architecture in general and the Semantic Web. These differences present a hindrance to fully leveraging the data-modeling innovations intrinsic to the hypergraph and Semantic Web paradigms, particularly in the context of multi-institution data integration — even though such integration is a primary motivation for the paradigms as outlined above.

Creating a Truly Unified Data-Sharing Framework

Against this backdrop, Linguistic Technology Systems proposes a new "Hypergraph Data Modeling" (**HGDM**) protocol, whose goal is to unify the principal structures of multiple hypergraph and Semantic Web frameworks, yielding a truly general-purpose data-sharing system. The **HGDM** protocol is focused on research-oriented communications between biomedical institutions.[2] **HGDM** aims to compensate for limited industry adoption of hypergraph-database and (to some extent) Semantic Web technology — not by deploying hypergraph database engines in place of legacy systems, but by facilitating the implementation of software layers which allow information spaces to mimic the behavior of hypergraph database engines with respect to multisite data sharing. That is, **HGDM** *first and foremost* defines a protocol whose canonical use-case is one of remote applications accessing hypergraph database content via a semantically-constrained network.[3] However, the protocol only stipulates a contract between server and client software *applications*, without explicit requirements on the server's physical data storage. An **HGDM** server could therefore be a software layer adapting filesystems or database instances of different kinds, and not only hypergraphs. This need not entail programming encompassing the entirety of an institution's local data; instead, developers could selectively curate information fitting some constrained criteria, as part of a targeted data-sharing project.

In short, **HGDM** governs data-sharing activity coordinated between three different software entities (*see* Figure 1). That is, a "server" (which is not necessarily a **TCP** server in the conventional internet sense) responds to requests for information against some data space (potentially but not necessarily a hypergraph database instance). After pulling relevant content from this data space, the server serializes the response data and sends it to an intermediate software component, which parses the response into a hypergraph data structure. **HGDM** introduces a novel "Hypergraph Exchange Format" (**HGXF**) for encoding/serializing hypergraph data. The intermediate software, as such, then receives **HGXF**-encoded data and parses this content to create an "infoset," similar in purpose to an **XML** post-processing infoset. **HGXF** utilizes an extended version of the **TAGML** markup language[4] for a hypergraph-based document syntax, and derives a hypergraph-based semantics from the **HGDM** infoset protocol. The information contained in this infoset is then translated into "application-level" objects — viz., instances of data types which are natively recognized or implemented by the client application that initially formulated the **HGDM** request. Accordingly, the third end-point for **HGDM** components would be a library embedded in applications, one which implements functionality to initiate **HGDM** requests and can translate **HGXF** response-data infosets into application-specific data.

The **HGDM** protocol establishes guidelines or requirements for each of these three data-sharing endpoints/layers. The computational units within these components will be generally refered to, below, as "objects," so that we have *server objects*, *infoset objects*, and *client objects*. **HGDM** regulates communications between server, infoset, and client components by stipulating or recommending that these objects (and the data types of which they are instances) support specific

[2] There are actually no technical details in **HGDM** which restrict it to a clinical, health-care, or bioinformatic context, so implementations could potentially be beneficial in other sectors.

[3] According to semantics defined by the protocol: hypergraph structures are embraced as representations for content shared between applications, whether or not the communicating end-points adopt hypergraphs natively.

[4] See https://pdfs.semanticscholar.org/bbd9/9215f6bed393c9274f8e0642bebf42d8f633.pdf.

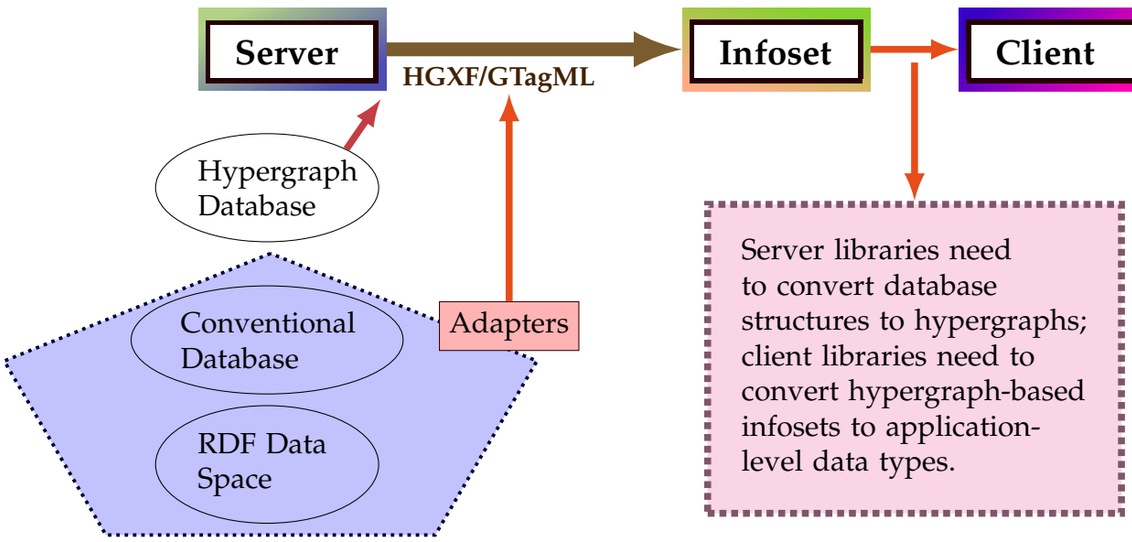

Figure 1: Outline of HGDM Server, Infoset, and Client Layers

methods and procedures, many of which are oriented toward sustaining hypergraph-based data models across each phase in the data-sharing process.[5]

The following sections will outline some of the procedures standardized for the three protocol layers (server, infoset, and client) and will also address some details concerning how data within the hypergraphs should be encoded.

Protocol Outline for Server, Infoset, and Client Objects

As indicated in the Introduction, there are three different contexts or stages where **HGDM** components would be implemented: *servers*, *infosets* (objects parsed directly from serializations), and *client* applications (values used directly by user-facing software components, which are initialized from objects of the prior two kinds). Server objects may be directly obtained from a hypergraph database, or alternatively from "adapters" (components that access non-hypergraph databases in a manner which emulates hypergraphs). The server, infoset, and client components will sometimes be referred to as "layers" or "endpoints" (considering that each is potentially the endpoint of a network communication governed by **HGDM**). Objects within each of these three layers can be conceptualized as *hypernodes*, meaning that they have both internal structure and labeled connections with other objects; although these hypernode structures are completely formalized only at the infoset layer.

The material below provides a summary of **HGDM** by enumerating certain procedures within the protocol specifications for each layer. The procedures are presented as grouped according to the layer where they are most directly relevant; however, objects may also implement procedures from other layers. Of the three endpoints, the intermediate "infoset" layer is the one most strictly regulated by **HGDM**. By contrast, the server and client components have more latitude, recognizing that these layers may be implemented in the context of disparate application and data-persistence environments. Nevertheless, it is consistent with **HGDM** for the server and client layers to implement procedures and data structures mimicking the hypergraph constructions of the infoset layer.

[5] According to the goal of representing, at a minimum, the important structural contributions of major hypergraph database engines, one question then arises as to how to identify such engines as a subset of overall database technology. Considering a range of commercial and academic projects, **HGDM** draws from (in particular) **HyperGraphDB**, **Grakn.ai**, AtomSpace (see https://aiatadams.files.wordpress.com/2016/01/cogprime-overview.pdf), **LMNtal** (see https://waseda.pure.elsevier.com/en/publications/lmntal-a-language-model-with-links-and-membranes), **Neo4j** (see http://www.we-yun.com/doc/books/Neo4j%20in%20Action.pdf or https://academic.oup.com/nar/article/48/D1/D344/5580911), and **WhiteDB** (see http://whitedb.org/pr https://arxiv.org/pdf/1910.09017.pdf), as well as several runtime (non-persistent) hypergraph libraries.

Server-Layer Procedures

This subsection will review procedures associated with server objects. Some of these procedures are associated with specific database architectures, so therefore they may not be implemented in every HGDM server-layer component — although providing capabilities concordant with multiple database architectures allows components to serve as adapters integrating various databases into an HGDM network; as such, these procedures can be seen as a guideline for integration across heterogeneous information systems.

get_binary_encoding() Provides either a raw byte array or base-32 encoded byte array, which is a binary serialization of a server object. This encoding should be reversible, in that a procedure exists to recreate the serialized object through the byte array.

HGDM recommends a base-32 encoding scheme designed to be compatible with **QString** lexical casts (note that this encoding differs from other base-32 systems commonly used).[6] Base-32 streams in HGDM are case-insensitive, little-endian, and use digits **0**-**9** and letters **a**-**v**. The additional alphanumeric characters **w**, **x**, **y**, **z**, and underscore ("_") are employed as end-of-stream separators/markers, indicating 0-4 padding bytes.

get_decoder_class() Returns the name of a class whose objects are binary-encoded (and consequently decoded) by **get_binary_encoding**.

get_decoder_procedure() Returns/provides a data structure through which one obtains a procedure constructing decoder-class objects from byte arrays.

get_indexes() Returns a list of fields within a server object (construed as a hypernode) which are indexed for querying, meaning that one can identify individual objects based on the value for that field (and also potentially sort a collection of objects via that field).

Borrowing terminology from HyperGraphDB, HGDM refers to the smaller units within hypernodes as "projections."[7] At the infoset layer, all objects are fully-formed hypernodes wherein all data contained by each object is available within "hyponodes" — the nodes inside hypernodes; the hypernode is said to *project onto* each of its hyponodes. In the server-layer context, using a design analogous to HyperGraphDB, only some of the relevant information encoded via an object may be available via projections (the remainder is binary-encoded and therefore not accessible to query evaluators outside the object itself). In general, the data fields which have projections within server objects are those that have indices.

get_table_structure() Returns/provides a description of how tabular data can be mapped to a collection of hypernodes.

This procedure is associated with adapters for SQL-style databases. In general, data layouts which are technically distinct from the hypergraph perspective may all be embodied by relational tables in the SQL paradigm, obscuring the hypergraph-structural variations. This procedure compensates for that ambiguity by providing an outline of how a given table should be mapped to hypergraph formations. In the simplest mapping, each record becomes one hypernode, and each column corresponds to one projection. However, more complex mappings are also possible. For example, one column in the table might hold map keys, while the other columns are aggregated into hypernodes, so that the overall data structure is treated as a map (associative array) whose values are hypernodes. Or, columns may be aggregated into two sets

[6] **QString** refers to the principle character string/text class of the Qt C++ application development framework. The encoding is structured so that base-32 strings can be mapped to decimal or hexadecimal numbers via **QString** methods such as **.toInt(...)**.

[7] See http://www.hypergraphdb.org/docs/hypergraphdb.pdf.

of hypernodes, with the overall data structure treated as a graph asserting relations between pairs of hypernodes selected from those sets.

get_rdf_mapping() Returns/provides a data structure encapsulating how **RDF**-style graph data may be interpreted in a hypergraph context. This procedure is associated in particular with adapters working with Semantic Web services.

In general, mapping Semantic Web or **RDF** ontologies to hypergraphs is non-trivial (certainly too complex for an automated process or a single algorithm), because there are numerous sorts of inter-node relations in a hypergraph context which may all be embodied as generic graph-edges in **RDF**. Therefore, adapter schemes for **RDF**-style data should model how **RDF** relations translate to hypergraph constructions — including hypernode-to-hypernode connections, hypernode-to-hyponode projections, hyponode-to-hypernode proxies, and various combinations of these links.

get_file_type_description(), get_file_type_class(), get_file_type_decoder() These procedures are principally intended for use when adapters encapsulate access to a local filesystem, or individual files therein, in accordance with a hypergraph protocol.

Infoset-Object Procedures

In **HGDM**, infoset objects are hypernodes containing sequences of individual parts (hyponodes), and also connected to other hypernodes via directed edges, or "connections." Each connection has a "connector" object which is itself a hypernode, but is not (directly) connected to other hypernodes, apart from the source-connector-target triple. All hypernodes and hyponodes have a type, and the hypernode type constrains the type of its hyponodes.

In **HGDM**, hypernodes may have a dynamically changing sequence of hyponodes — the length of this sequence may enlarge or contract. However, new hyponodes must be added according to a fixed type pattern. To be precise, all hypernodes have a (possibly empty) list of *structure fields*, which are hyponodes each of which is associated with a predetermined type and a string label. Each of these fields must have a value (perhaps a "null" value if such a value exists in the hyponode type); and hyponodes cannot be added or removed from the structure-field part of a hypernode. In addition to (and sequentially following) the structure fields, hypernodes may have *array fields*, which in the simplest case is zero or more hyponodes all with the same type. Within the span of the array fields, hyponodes may be added to or removed from the sequence. **HGDM** permits array fields to have multiple types, but in this case the types must conform to a predictable "cycle," meaning that every nth hyponode (for some n) has the same type. For instance, a map between strings and integers can be represented by a length-two cycle alternating between string-typed hyponodes and integer-typed hyponodes.

Many of the canonical **HGDM** procedures in the infoset context involve hypernodes, hyponodes, array fields, and structure fields. These include:

get_hypernode_type(...), get_hyponode_type(...) Returns a name or a data structure representing the corresponding node's type.

In general, each hypernode's type determines both the label used to access items in its sequence of hyponodes, and also the corresponding hyponode's type (note that only structure fields have string labels, however).

get_structure_field(...), get_structure_fields(...) Provides access to one or more of a hypernode's projections (i.e., its contained hyponodes). Each hyponode may be obtained via an index into the containing hypernode's hyponode-sequence, but using a descriptive label is more

portable and self-explanatory than using raw indices. Variants on these procedures which take multiple labels will return or expose multiple projections accordingly.

HGDM recommends that access to projections be provided via a "functional" style as well as (or in place of) returning a handle to hyponodes directly. This means that the procedures have variants which accept a procedural value (e.g., a code block) which the called procedure will execute (assuming a proper hyponode can be provided), passing the relevant hyponode as a value to the code passed as an argument. If multiple hyponodes are requested, this code should be called multiple times, one for each hyponode.

get_array_field(...), get_array_fields(...) Provides access to one or more hyponodes from the "cyclic" part of the hypernode.

Array fields are one-indexed (meaning that the field corresponding to the number "1" is the first field in the cycle, and so forth). Note that the array index is not (in general) the same as the index in the overall hyponode sequence.

Even if the primary role of a hypernode is to hold a collection of values — i.e., a resizable array whose hyponodes are indexed via array fields — hypernodes may pair these arrays with one or more structure fields carrying holistic info about the array. **HGDM** stipulates that hypernode implementations should indicate the length of a hypernode's total hyponode sequence; the boundary between the structure fields and array fields; and the "cycle length," which derives from the recurring type-pattern governing how hyponodes may be added as array fields. However, depending on how hypernodes encode their data, the array fields may embody a "logical" sequence of implicit values spread over multiple hyponodes. In some of these cases the number of distinct logical values encoded within a hypernode cannot be determined just from sequence- and cycle-length data alone. This is an example of the sort of holistic data which might reasonably be stored in structure fields preceding an array-field cycle.

As with structure fields, array-field access can be provided through functional variants, and through variants which return/expose multiple hyponodes. Variants should be provided which expose multiple array fields both via a list of indices and via a range of indices, indicating interest in each field indexed between the range start and end (inclusive).

get_field(...), get_fields(...) These are lower-level procedures which expose fields based on raw (zero-indexed) sequence numbers. These procedures do not distinguish whether the accessed hyponode is a structure field or an array field.

get_role_projection(...), get_conceptual_role_structure(...) Exposes a hyponode via a description of the "role" played by that value in a hypernode, which is not (necessarily) the same as a structure-field label.

Following **Grakn.ai**, **HGDM** recognizes that some hypernodes are representations of events or situations which comprise multiple parts or "actors" playing different roles.[8] Labeling these roles provides an alternative interface for accessing the hyponodes onto which the roles project, in the context of a given hypernode. These projections may potentially be array fields as well as structure fields, particularly when the same role is played by multiple parties.

HGDM allows this concept of "roles" to be generalized to role-aggregates in a manner which borrows from Conceptual Space theory as well as conceptual-role semantics, which motivates the **get_conceptual_role_structure(...)** variant. Conceptual Space representations are also relevant to the next item.

[8] See http://klarman.me/academic-work/resources/MesEtAlCISIS17.pdf.

get_dimension_structure(...), get_domain_structure(...), get_conceptual_domain_structure(...), get_hybrid_conceptual_structure(...)

Returns/provides dimensional information about a projection, including numerical/statistical detail such as level of measurement (Nominal, Ordinal, Interval, Ratio), units of measurement, scale, ranges, or expected distribution within a range. The **HGDM** semantics of dimensions (and domains, which are dimension aggregates) is derived from the Conceptual Space Markup Language **CSML**.[9]

In accord with Conceptual Space theory, dimensions may be grouped into *domains*, and domains grouped into "concepts." These aggregative structures can be defined or indicated via **get_domain_structure** and **get_conceptual_domain_structure**. **HGDM** also allows domains to be characterized in terms of roles as well as dimensions; as such, **get_hybrid_ conceptual_structure(...)** yields a structural articulation which combines the features/details of **get_conceptual_** and **get_conceptual_role_structure(...)**.

get_connection(...) Given a hypernode (the source) and a description of a hypernode attached to a hyperedge (the connector), returns a hypernode which is the target of the edge whose source and connector matches the arguments.

If multiple edges match, and therefore multiple targets, the procedure can either return all matching hypernodes or else (assuming it is possible to order the targets) the first target which matches.

HGDM has a notion of *frames* and of *contexts* which might constrain connection matches. When contexts are in effect, certain connections (i.e., connections between hypernodes) may only be considered applicable when one or more contexts are "active." Similarly, "frames" are sets of hypernodes and/or hyperedges, and a certain connection may "exist" within one frame but not outside it. When frames and/or contexts could be in place which filter connections, any **get_connection(...)** procedure should take frames or contexts as parameters, and attempt to match edges within the confines of those structures.

get_proxy(...) Returns a hypernode which is the target of a hyponode "proxy." Proxying means that a hyponode has a *value* which designates or points to a hypernode. **HGDM** uses proxies as an alternative form of linkage between hypernodes — via one hyponode which holds the proxy value — in contrast to connection-based hyperedges.

consruct_hypernode_via_proxies(...) Given a sequence of hypernodes, constructs a new hypernode whose projections are proxies to each of those hypernodes, in turn. A variation of this procedure should be one which also takes a pre-existing hypernode as a parameter, and appends the proxies as array fields to that hypernode.

Note that for each hypernode type, proxies to that type represents a distinct type applicable to hyponodes. Hypernodes can only have fields which are proxies if those proxy types are part of the hypernode's type profile.

get_binary_proxy(...) The notion of *binary proxies* is available in an **HGDM** context if it is possible to encode sequences of hyponode values to a compact binary representation. In that case, it is possible to indirectly encode the data within a hyponode-set as a byte array. One form of **get_binary_proxy** should therefore construct a hypernode whose projections are translated

[9] See https://opus.lib.uts.edu.au/bitstream/10453/31756/1/2013007531OK2.pdf, https://www.researchgate.net/publication/221406067_Conceptual_Space_Markup_Language_CSML_Towards_the_cognitive_SemanticWeb, https://www.semanticscholar.org/paper/A-Metric-Conceptual-Space-Algebra-Adams-Raubal/521acbab9658df27acd9f40bba2b9445f75d681c, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.302.8106&rep=rep1&type=pdf, http://ceur-ws.org/Vol-2090/paper4.pdf, or https://arxiv.org/pdf/1703.08314.pdf.

from such a binary encoding. Note that the resulting hypernode is not considered to be part of the graph — it is not connected to any hypernode via either hypernode connections or actual proxies.

A variation on this construction combines the properties of bonafide proxies and binary proxies. In this case, a hyponode contains both a proxy value and a binary value compactly encoding that proxied hypernode's content. When the proxied hypernode changes, the binary proxy is automatically updated.

Whether used alone or paired with a bonafide proxy, binary-proxy values may be part of a union type (i.e., a type which can be declared which *either* holds a binary proxy *or* some other value, such as an integer).

get_channel_structure(...) In **HGDM**, a *channel* is in effect a special form of frame which aggregates some connectors that share a target. Channels may have labels, and be regulated by a *channel system* which identifies what sort of channels may be applied and whether there are restrictions on how they are combined (e.g., whether a non-empty channel — of a given kind — leading toward one hypernode precludes, or alternatively requires, an additional non-empty channel leading to the same hypernode). The **get_channel_structure** procedure should, given a target node, identify the set of source nodes which connect to the target, grouped by channel.

Note that efficiently finding source nodes from target nodes is only possible in general if the **HGDM** software implements "reversible" connections, meaning that target-to-source relations are stored alongside source-to-target. Whether or not such reverse-connections are modeled in general, they should always be provided once connections are associated with channels (i.e., reverse edges should be represented as a data structure attached to channels' target hypernodes).

get_hyponode_set(...) Given a hyperedge, returns a sequence of hyponodes comprising all hyponodes in the source hypernode and then the target hypernode, effectively erasing the boundary between the two hypernodes in the edge context. This is an example of mapping hyperedges to hypernodes, which is a basic operation of hypergraph analysis.

Client-Object Procedures

In general, application-level objects are initialized from **HGDM** infoset objects, whose hypergraph-based structures are designed to encode many data structures in a flexible, expressive fashion. Client objects, in contrast to infosets, are not intended primarily for sharing between disparate applications; as such, these objects can be modeled directly according to the needs of individual applications, in terms of building compelling **GUI** front-ends and effective domain-specific analytic capabilities. Constructions which are specific to hypergraphs — e.g., role-projections, proxies, and inter-hypernode connections — are not necessarily appropriate in the application-level context (often they can be replaced by ordinary application-level type structures, such as pointers and object members with **get**/**set** accessors).

Nevertheless, the **HGDM** protocol recommends several design patterns or practices which can be enacted at the application level to help application code translate infosets to local type-instances, and in general to interoperate effectively with **HGDM** networks. These patterns are particularly applicable to "collections" types (the sorts of values encoded/decoded at the infoset layer via array fields) and to string/textual data (particularly in contexts where words or names from multiple languages may be involved).

With respect to data types considered in most contexts to be "atomic" values — e.g., strings, integers, and decimals — **HGDM** encourages developers to use types and encodings that transparently document scientific and implementational assumptions and requirements. For instance,

magnitudes can be decorated with dimension units and range-restrictions. Floating-point values can be expressed with extra precision by employing ratios (integer quotient-pairs) or "posits" (a special number system invented by John Gustafson)[10] in lieu of traditionally-encoded floating-point representations.

With respect to non-atomic "struct" types, whose contents are accessed by named data fields, **HGDM** recommends accessors in which the bare field-name serves as a get-accessor and the form "**set_**" provides setters. Procedures named "**get_...**" should not be used for accessors (which logically serve merely to return some member data) but for obtaining a value which requires some intermediate computation, or a value which may vary according to some context beyond the prior explicit setting of one single member value.

In terms of non-atomic collections types, **HGDM** recommends a collections protocol which overlaps with some patterns applicable to infoset array fields. In particular, **HGDM** recommends collections types whose underlying sequential data is one-indexed, although with a "zero" value allocated in memory to represent "default" or "null" values. Internally, that is, the sequences are zero-based — usually it is not necessary to modify access indices — but the actual value present at position zero is considered not a *logical* part of the sequence, but rather a placeholder for a value to use for out-bounds or other exceptional circumstances. If **arr** is an array and n is an integer, the expression **arr[n]** would then return **arr[0]** for any n greater than **arr**'s size.

This system addresses several suboptimal properties of both zero-indexed and one-indexed arrays. On the one hand, there is a correspondence between the length of an array and the index of it's last element: if n is the length of **arr**, then **arr[n]** is the last element in **arr**. Moreover, if one searches for an element in **arr** which is not found, the return value can be naturally **0**, numerically the same as boolean **false** — and not (as is common practice in zero-indexed environments) **-1**. Employing **-1** as a "sentinel" not-found value is problematic in several ways, one of which is that **-1** evaluates under boolean conversion to ***true*** rather than **false**. Indeed, **-1** does not even technically exist if the array index is an unsigned integer. Moreover, some systems allow negative indices to support counting from the *end* of the sequence — **arr[-1]** would be **arr**'s last element, **arr[-2]** its next-to-last, etc. In this case it is dubious to use **-1** both as a valid index value and as a flag for a index not found — i.e., for the *lack* of any index which returns a value that would satisfy some condition. With any logic, the proper flag value for a "not-found" condition would be zero. On the other hand, computationally offsetting every index by one is at least a little inefficient and potentially error-prone. A reasonable compromise is to leave an array internally zero-indexed, but treat the first (zero-index) value as a "dummy" value which is not logically part of the array. In that case, then, one can utilize this initial value as a default to be provided in out-of-bounds situations. With that practice, **arr[n]** becomes defined for all n — defaulting to **arr[0]** for n out of the proper index range.

The **HGDM** protocol recommends employing such a one-indexed system as a basis for multiple collections types that may be based internally on arrays, including lists, stacks, queues, and even matrices (two-dimensional arrays). **HGDM** also recommends providing "functional" accessors to collections types (that is, implementing variants of procedures which call a passed code-block with a value obtained from a collection, or potentially multiple calls with multiple values, in lieu of return values directly). Other guidelines, particularly those related to textual data, will be addressed in a later section.

at(...), get(...), value(...), get_at(...) These procedures provide indexed-based access to collections which support access at any index (not just the start or end of a list). The first element is

[10] See http://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf.

at position **1**.

When implementing indexed-based access, the basic question to address is how to handle out-of-range indices. These distinct procedures use different strategies for such cases. Specifically, **value** (similar to **Qt**'s collections) requires a default value to be provided for out-of-bounds indices (a default-constructed value may be used for types which have one, so this default value would often not be explicitly passed as an argument). The **at** variant is recommended to return a reference to a value, defaulting to the "dummy" zeroth value if necessary. The **get** variant returns a pointer to a value, or a **null** pointer for out-of-bounds indices. Finally, **get_at** can mimic the behavior of either **at** or **value**, but in either case **get_at** should provide a method syntax for the same operation achieved by overloading square brackets: i.e., **arr[n]** should be the same as **arr.get_at(n)**.

Note that programmers seem to differ on whether (in an ideal language) bracket access (viz., **arr[n]**) should return a (copied) value, a mutable reference, or a constant reference. **HGDM** allows for each of those options, to accommodate different computing environments. One note, however: in a **C++** context, it is possible to mix all three styles by an expanded implementation of **operator []**. The simple step here is using **const**/non-**const** overloading to provide two different forms of the procedure, one returning a constant reference and one a mutable reference. The more complex step (and one which some developers might deem too much a departure from idiomatic **C++**) is to associate **operator []** with an intermediate *functor* type which has a *cast* operator to a reference but a *call* operator to a value, via a provided default value. The details of this implementation are outside the scope of this summary, but the upshot is that with operator-overloading along these lines default values can be provided even with bracket notation via code such as **arr[n](1)** — observe the trailing "**(1)**" here which indicates that the number **1** should be used as a default if n is out of range.

push(...), pop(), top() These procedures apply to collections behaving like stacks — "Last In, First Out" data structures in which the element most recently added to the collection is removed by default, when no other criteria for an element to be removed is provided. All stack-like types should provide **push** (add an element to the stack), **top** (get the most recently added element), and **pop** (remove the top element). Stacks *may* also provide procedures to access and append/remove elements in other positions as well, if the stack can also be used as a different kind of collection; however, **push**, **pop**, and **top** are the characteristic functions associated with stacks.

enqueue(...), dequeue(), head() These procedures apply to collections behaving in the manner of queues — "First In, First Out" data structures in which items that have been present in the collection longest are removed first, by default. Every item added to the queue is placed logically "behind" the prior elements. This kind of collection is visualized like a line-up of objects, rather than as a vertical "stack," which explains the common terminology used for queue-oriented procedures rather than stack procedures. As with stacks, elements *may* be accessed out-of-turn, but **enqueue**, **dequeue**, and **head** are canonical functions associated with queues.

push(...), enlist(...), pop(), delist(), top(), rear() **HGDM**'s terminology for deques (double-ended queues) differs from conventional procedure-names more so than for stacks or queues. **HGDM** considers deques to be a combination of two stacks, one representing the "front" of a collection and one representing the "back." Deques in general are queues in which elements may be added to the front — where they are the first to be removed — as well as to the rear; in effect, an algorithm can select to allow some element to "skip the queue" when it is added. Logically, this has the effect of forming a "front" and a "back" stack, where the former is en-

larged whenever an element is added which may "skip the queue" and where the latter is enlarged whenever an element is added in conventional "First In, First Out" fashion.[11]

HGDM accordingly uses non-standard procedure-names for the "back" stack, intended to convey the stack-like behavior: **enlist** adds an item to the rear, **delist** removes the item at the rear, and **rear** returns this item. Note that an item added via **enlist** may eventually be removed via **pop** (if all items ahead of it are removed first); however **delist** removes the rear-most item *first* (similarly, an item added via **push** might eventually be removed via **delist**).

is_row_major(), is_column_major(), is_diagonal(), is_symmetric(), is_skew_symmetric(), get_matrix_length(), get_matrix_size()

These procedures are applicable to matrices (or two-dimensional arrays), which in **HGDM** are assumed to be represented internally as a one-dimensional array. Most numerical or linear-algebra libraries adopt this convention, translating two-dimensional indices to a one-dimensional index into an internal array. Matrices in *row major* order store each row in contiguous memory, so the whole first role is at the beginning of the internal array, then the whole second row, etc. Conversely, matrices in *column major* order store columns in contiguous memory. Libraries differ in whether they employ row-major or column-major order, so **HGDM** recommends matrix classes support *both* options, to minimize memory-copying when importing data from other contexts. In general, row-major order is more efficient when it is desired to copy or reference individual rows, and column-major is better for copying or referencing individual columns.

Note that some matrices are neither row-major nor column-major. In particular, diagonal, symmetric, and skew-symmetric matrices have redundancies which make it unnecessary to store every item on its own. For instance, for diagonal matrices it is only necessary to have an array which is the length of the diagonal; all other elements (i.e., positions r, c for $r \neq c$) are zero. When the matrix in these cases is represented in a manner which minimizes the length of the internal array, neither rows nor columns are contiguous in memory.

Note also that these cases reveal a divergence between the logical number of elements in the matrix (the product of the number or rows and number of columns) with the length of the internal array. As such, **HGDM** recommends distinguishing matrix *size* (logical size) from matrix *length* (the actual array length). For instance, for a typical square diagonal matrix with n rows/columns, **get_matrix_length** would return n, while **get_matrix_size** would return n^2.

get_dimension_structure(), get_column_dimension_structure(), get_pseudo_constructor() Except where computing speed has to be strictly optimized, **HGDM** recommends using more detailed data types — even for numeric values — than conventional integers or floating-point decimals. For natural numbers, range and/or unit-restricted types ensure that values are used in scientifically reasonable ways. For a collections type, **get_dimension_structure** and related procedures would then return information about the range, scale, and units of measurement applicable to all elements in the collection. Along similar lines, **get_column_dimension_structure** would provide this information relative to an individual matrix column, in a situation where different columns could potentially have different dimensional details.

For each instance of a range-bound numeric type, **HGDM** also recommends proving a static factory function (a "pseudo" constructor) which confirms that the requested initial value is in the allowable range before calling an actual constructor. A pointer to that function — which may be obtained via a templated **get_pseudo_constructor** procedure — may then serve as a "passkey" value signaling that the provided constructor argument lies within the requisite range.

[11] It is possible to *literally* implement queues as double-stacks, but one must then track the position where the two stacks meet and accommodate the possibility that one stack may be empty.

The prior sections in this paper have considered hypergraph-based information encoding/serialization with an emphasis on numeric and collections types, recommending representational practices which document scientific and programming assumptions and requirements. The underlying principle — that data encoding should be rigorous and transparent so as to clarify the data's scientific background — also applies to textual data, implying the need for carefully designed text-representation protocols.

Properly encoding textual, Natural Language content is one of the more complex and challenging aspects of rigorous data sharing. The Unicode standard theoretically provides a consistent framework for representing all the world's languages, even under-resourced languages (plus many other symbols, e.g. those used in mathematics). However, there are several different Unicode encodings, with no guarantees that two separate communicating applications would employ compatible encodings. Unicode also has certain representational limitations (outlined below), so it is not a definitive textualization solution.

Most textual data needs fewer than 256 distinct characters — the most that can be represented with one byte — even if the text contains some special symbols, such as accented letters occurring in foreign names. It therefore wastes a lot of memory to allocate more than one byte per character, especially for long text documents. However, *which* 256 characters are needed can easily vary from one text to another. Generic types holding textual data, such as **QString**s, often employ two bytes per character instead, for greater flexibility. One limitation of this solution, however — apart from extra memory-usage — is that characters cannot be paired one-to-one with two-byte units: in Qt, for instance, some Unicode glyphs actually require *two* **QChar**s in sequence. The full Unicode system utilizes a combination of one-byte, two-byte, and four-byte glyphs; as such, there is no correlation between the index of a glyph in a character stream and the position of a byte in a corresponding Unicode array. To retrieve the nth character in an unrestricted Unicode text, for example, it may be necessary to examine every prior character so as to determine which byte(s) actually holds the character requested. A further consideration is that users often want to copy-and-paste across applications; given that distinct applications often employ incompatible character-encoding schemes, a thorough encoding system should support the notion of a "copyable" range which can convert the indexed characters to a neutral format.

In short, efficiently indicating individual characters or character-sequences by positional indices is oftentimes more important than strict adherence to Unicode (or any other) standards. For example, Natural Language text may need to be annotated wherein mentions of names or concepts are identified via index-ranges. In a biomedical context, strings representing proper names (e.g. patients or doctors), diagnostic codes, symptoms, clinical procedures, chemical formulae, and so forth, may all be detected via text mining (Named Entity Recognition, Lemmatization, Diarization, etc.). The representation of such annotation data could become especially complicated without a straightforward indexing scheme — that is, without there being at least one view onto a document such that the overall textual content is a list of characters, with text segments identified by providing a start and end index. For efficient indexing, every character should accordingly have *the same number* of bytes.

The technical question therefore emerges of how to flexibly encode a variety of characters (including certain rare glyphs only used once or twice in a document) while enforcing that all glyphs span the same length of bytes (preferably just one byte) and also preventing unnecessary wasted memory. There are two possible (but not exclusive) solutions to this problem. One option is to reserve a certain number of character codes which are not assigned, except for within individual documents — allowing them to locally map those codes to a more expansive char-

acter set, such as Unicode. Another option is to maintain a separate index — outside of the principal character array — for the handful of special glyphs which a document may need outside its normal character set. For instance, suppose three characters out of several thousand in a document require glyphs that do not fit within a one-byte character set. This document could use a special code (e.g., **255**) for these characters in the main array and then, separately, maintain a mapping from the three index points to (say) Unicode glyphs intended to be inserted into those three positions.

Sometimes these strategies can be combined. For example, many non-English letters appearing in English-language texts are accented characters from other European languages, belonging to proper names or special phrases (cf. the "à" in *vis-à-vis*). These glyphs are typically described by providing a "base" letter plus a "diacritic" (accent) mark. In a one-byte-per-character encoding, one could place only the diacritic code in the main character array, and use a separate mapping to fill in those letters (e.g. a data structure which would assert that the acute accent at position 102, say, should be placed atop an "e"). Only those character codes which could potentially need supplemental bytes (e.g. diacritic codes) would trigger lookup in that external mapping. Such a system combines the ideas of having a designated subset of codes for "locally defined" nonstandard glyphs and of having an indexed-based lookup for nonstandard glyphs at specific index-positions in a character array.

Aside from encoding limitations of Unicode and other popular character formats (including **ASCII**), another problem with these existing encodings is that they tend to address the visual appearance of glyphs rather than their semantic meaning. Consider an ordinary period, which could have several different interpretations: the end of a sentence, part of an abbreviation, the decimal point in a printed number, part of an ellipses, or an operator in computer code. Failure to encode such differences can make text mining more complex than necessary — for instance, using separate character codes for end-of-sentence punctuation than for other uses of periods, exclamations, and question marks would eliminate the need for **AI**-driven "Sentence Boundary Detection." In some contexts, differences in glyphs' linguistic meaning may translate to visual typesetting as well: periods marking an end of sentence, say, should be followed by larger space-gaps than those inside abbreviations (e.g., "Dr."). Hyphens are often printed with different lengths when used as punctuation (dashes) as opposed to within words or phrases (cf. *vis-à-vis* again) and within numeric literals (cf. "-1"). Quote marks are typically typeset as curved indicators when actually used to surround quotes, but as straight marks when used for feet and inches ("6'2" tall"). Such presentational details may only be considered in depth at the camera-ready phase of a scholarly publication, but in fact there is an overlap between the proper visual cues expressing certain punctuation or other non-alphabetic marks and their semantic meaning, which is relevant for text mining. As such, these semantic differences should optimally be encoded in character sets themselves (rather than relegated to special markup, such as LaTeX commands).

For a further consideration, the basic Latin-1 character set is also inefficient in including certain obsolete **ASCII** codes (such as the "bell" character, which literally rings a bell via the computer's audio) that are almost never currently used. There are, in short, *de facto* unassigned code points, leaving room for duplicate glyphs which may have the same appearance but alternate interpretations (e.g. abbreviation-period vs. punctuation-period). Combining all of these ideas, then, **HGDM**'s recommended text-encoding protocol — dubbed **HTXN** ("Hypergraph Text Encoding") — uses diacritics, "interpretation codes" (disambiguating semantically distinct but visually similar glyphs) and the option to extend character sets arbitrarily for individual documents, to yield a flexible and expressive encoding. This encoding, in its raw form, requires more than one byte per character, but **HTXN** employs certain tricks to compress the character set so

that it can typically fit into a single-byte context.

Character Encoding and TAGML

Having discussed the encoding of individual characters, it is appropriate to address **HGDM** recommendations for encoding markup and presentation details, at a higher scale than individual glyphs. As a general-purpose serialization format, **HGDM** suggests **TAGML**, which is similar to **XML** but allows certain constructions that are more expressive than **XML** (e.g., concurrent markup). However, **HGDM** describes a modified version of **TAGML**, one engineered to work with **HTXN** at the character-encoding level.

Intrinsically, **TAGML** does not specifically address character encoding; instead, this is assumed to be a property of each "node" which holds character data. In **TAGML**, certain nodes (called "prenodes" in **HGDM**) represent character-sequences, while other hypernodes embody markup, applied to groups of prenodes. The overall character sequence of a document is assumed to be retrieved by collating characters from every prenode, in order. This can be inefficient when it is necessary to define annotations, or other index-based ranges, which may span multiple prenodes (one cannot readily map single index numbers — assuming these quantify over a whole document — to character indices within individual prenodes). For this reason, **HGDM** recommends that a **TAGML** runtime hold character data outside of prenodes proper — specifically, that characters be stored in "text blocks" and prenodes use indices into these blocks to establish their character data. Each text block, in turn, uses **HTXN** to encode characters via single bytes (different text blocks may use variations on the basic **HTXN** encoding).

Note that this setup does not operationally alter the **TAGML** structure: while a text block *may* encode an entire document via a character array that can be accessed outside any prenode, this is only one possible use-case. Absent specific designs to the contrary, it should be assumed that the definitive character data for any document is accessible only via prenodes, and the use of text blocks is merely a memory optimization to streamline the initialization of prenode contents. In particular, prenodes which have overlapping indices to the same block are not overlapping markup — instead, each prenode holds multiple copies of a given character string, which have no logical relation to one another. For example, a distinct text block could be set up to hold proper names that have foreign characters. Individual names could then be carried within a prenode by giving the start and end indices of the name in its containing block. If a name appears multiple times in a document, several prenodes might then hold duplicate data — the same indices against the same block. But these prenodes do not overlap; each is understood to have a logically isolated copy of that name, as if the characters in the name were stored internally in the prenode's character data without the use of an external text block as the in-memory storage.

Separate and apart from the character-encoding issues analyzed above, **HGDM** also proposes extending **TAGML** in ways consistent with the intended use of **TAGML** as a general hypergraph notation format — in short, as the markup language for **HGXF** (when hypergraph serialization is done via text documents). Specifically, **HGDM** proposes a variant dubbed "Grounded" **TAGML** (**GTAGML**) in which **TAGML** hypergraph structures are refined to notate hypergraph relations within serialized data, not only (as in the current **TAGML**) among markup elements. That is, **GTAGML** structures are "grounded" in hypergraph relations intrinsic to the data being conveyed by **GTAGML** documents. Further information about such "grounding" is outside the scope of this outline, but may be provided by Linguistic Technology Systems on request.