The MOSAIC Data-Set Explorer (MdsX): Initial Developer's Overview

The MOSAIC Data-Set Explorer (MdsX) is a suite of code libraries which can be used to build native, desktop-style applications for viewing data sets. An MdsX "data-set application" is an application customized and tailored to a particular data set, or a repository including multiple data sets. An MdsX *notebook* is a particular strategy for organizing data-set applications, or parts of data-set applications. In addition to serving as standalone applications in themselves, MdsX notebooks can be embedded in other applications; for instance, in scientific software.

is paired with MOSAIC *portal*, a code library for hosting data sets and publications. The MOSAIC portal code includes custom LaTeX commands for building annotated, indexed PDF files, and a custom PDF viewer which can read MOSAIC annotations and utilize their information to interoperate with data-set applications. Mosaic data-set applications can also customize this PDF viewer to add functionality specific to its data models and scientific subject-matter.

In general, MOSAIC applications are designed to be distributed in source-code fashion. They are, by default, written in C++ and based on the Qt application-development framework. Mosaic is designed so that sophisticated data-set applications can be built with few (or no) external dependencies apart from Qt itself. In the typical scenario, users would build and run MdsX applications inside the Qt Creator Integrated Development Environment (IDE). However, MdsX applications can also be configured so that they (or some functionality they provide) can be run from a command line — which allows them to participate in multi-application workflows — or bundled as plugins or source-code extensions into larger software components.

The MOSAIC portal is developed by Linguistic Technology Systems (LTS), who can develop and/or host data/publication repositories across disciplines (see contact details at the end of this paper for more information). One repository under development with MOSAIC is the "Cross-Disciplinary Repository for Covid-19 Research" (CR2), a collection of data sets related to Covid-19, that is paired with a forthcoming Elsevier volume, *Cross-Disciplinary Data Integration and Conceptual Space Models for Covid-19*. In addition to MOSAIC, LTS also provides a Dataset Creator (dsC), which is a QtCreator plugin helping researchers and programmers curate data sets and implement data-set applications.

Mosaic is designed to bridge the gap between scientific software and scientific data sets. While increasing volumes of open-access research data is becoming available to readers, researchers, and scientists, this data is not always published in a manner which facilitates reuse and interoperabil-

ity with scientific software — the kinds of applications that scientists themselves use to conduct and examine experiments or simulations. Moreover, the software-development ecosystem which is evolving around data publishing (as far as exchange protocols, file formats, development tools, and so forth) is methodologically removed from the engineering norms and principles of most scientific software. As such, a technical gap exists between the data-publishing and scientific-computing ecosystems seen as software-engineering domains. **Mosaic** aims to be a suite of tools which can help bridge that gap.

Data Sets and Data Publishing: the current picture

Recent years have seen an increasing emphasis, in the academic and scientific worlds, on *data publishing* — sharing research data and experimental results/protocols via web portals complementing those that host scientific papers. Published data sets now take a position alongside books and articles as primary publicly-accessible outputs of scientific projects. Coinciding with this increased volume of raw data, there has also emerged an ecosystem of tools allowing researchers to find, view, explore, and reuse data sets. These tools enhance the value of published data, because they decrease the amount of effort which scientists need to make use of data sets in productive ways.

Unfortunately, however, this ecosystem of tools does not include extensive work on software *applications* for accessing and using published data sets. Prominent publishes (Elsevier, Springer, Wiley, de Gruyter, etc.) have all developed suites of components for manipulating data sets and data/code repositories, including **APIs**, search portals, Semantic Web ontologies and other forms of Controlled Vocabularies, and cloud-based computing or visualization engines, founded on technologies such as Jupyter, Docker, and **WebGL**. However, none of these publishers actually provide *applications* for accessing data sets outside of the online resources where data sets are indexed. While these online portals can provide a basic overview of the data sets, publishers do not provide tools to help researchers rigorously use any data sets once they are downloaded. Moreover, the ecosystem for manipulating published research is largely disconnected from the software applications which scientists actually use to do research. The ability to work with data-publishing tools has not been implemented within most scientific-computing environments.

These lacunae may be explained in part by publishers' and scientists' hopes of creating cloud-hosted environments that can themselves serve as fully featured scientific-computing frameworks, with the ability to run code, evaluate queries, interactively display **2D** and **3D** graphics, and maintain user and session state so that researchers can suspend and resume their work at different times. In these cloud environments, users can run computations and generate complex graphics on remote processing units, with relatively little data or code-execution stored or performed on their own computers. Such employment of remote, virtual programming environments is sometimes necessary when interacting with extremely large data repositories; and can be a convenient way to explore data

sets in general, especially if a user is unsure whether or not a given data set is in fact germane to their research. Investigating data via cloud services spares the researcher from having to download the data set directly (along with the additional software and requirements which are often needed to make downloaded data functionally accessible). However, cloud-based data access is limited in important ways, which makes relying solely on cloud services to provide the filaments of a research-data ecosystm a very bad idea. The first problem is that cloud services are, despite their technical features, essentially just web applications under the hood; as such, they are susceptible to the same User Expericne degradation as any other web service — subpar performance due to network latency, poor connectivity, and the simple fact that web-based graphics can never be as responsive or as compelling as desktop software, which can interact directly with the local operating system and react instantaneously to user actions. The second, more serious problem is that could-computing environments are computationally and architecturally different than the native-application contexts where scientific software usually operates. Insofar as researchers develop new analytic techniques, implement new algorithms, or write custom code to process the data generated by a new experiment, these computational resources are usually formulated in a local-processing environment that cannot be translated, without extra effort, to the cloud.

To be sure, scientists can sometimes "package" their experimental and analytic methods into a coherent framework, such as a Jupyter notebook, which serves as both a demonstration and a precis of their research work. Indeed, tools such as Jupyter (which packages code, data, and graphics into a self-contained Python-based programming environment) are useful in part because the content shared via these systems (e.g. Jupyter "notebooks") needs to be deliberately curated; building a notebook is a kind of summarial follow-up to actual research work. The intellectual discipline involved in packaging up one's research via such tools may be a valuable stage in the scientific process, but even then the programming environment where research code and data is publicly shared is fundamentally different than the environment where the research is actually carried out. As a consequence, sharing research indirectly via cloud services and or "notebook"-oriented frameworks like Jupyter is not really conducive to either reuse or replication. To actually replicate a course of investigation, it is more thorough to employ the same (or at least functionally equivalent) software for data acquisition, analysis, and validation as the original software; and to incorporate published data in new projects, the data should be shared in such a way that the original research data, code, and protocols can be absorbed into a new research context, including the software used by the research team. Cloud-based services, which provide only an overview of research data, with limited analytic and imaging/visualization functionality compared to actual scientific software, do not substantially promote data replication and reuse insofar as these cloud services are functionally disconnected from scientific applications themselves.

This is the motivation behind x, a *native*, *desktop-style* application for accessing research data of different kinds — within the overall space of published data sets we can find specific variations, such as *data repositories* comprising multiple data sets; *image corpora* designed as test beds for Machine

Vision and diagnostic-imaging methods; *simulations* which involve not only raw data but digital experiments that can be re-run as a way to access the data; and so forth. Each of these various kinds of data sets present different sorts of interactive specifications which must be implemented by the data-set explorer software. While executed as a native application — not a cloud service — **MdsX** nevertheless incoporates the important ideas from contemporary data publishing (including ideas originating in the cloud context): workflow models, notebooks, access to publishers' **API**s, etc. Another feature of **MdsX** is that it can be run as a standalone application *or* embedded in other applications — such as the software which researchers are already using. In short, **MdsX** can be seen as akin to a cloud-based data-publishing platform where the "cloud" is replaced by a scientific-computing application. Instead of being hosted remotely ("on the cloud"), **MdsX** is hosted within a local desktop application. This host application may be pre-existing program, or a custom host implemented to allow **MdsX** data-sets to be explored in standalone fashion (with a default implementation that can, as desired, be modified for individual data sets/repositories).

The Structure of MdsX Notebooks

A common feature of software through which users study and reuse research data sets is some form of "interactive notebooks," or digital resources combining data, code, and graphics/visuals. The main feature of notebook-oriented design is the idea of interactive code editing, where changes in the code directly leads to changes in a visual display (such as a plot or diagram) which is viewed alongside the code. This setup allows developers to present or demonstrate data sets, and associated code, in an exploratory and interactive manner.

The exact details of how "notebooks" are designed and implemented varies between different technologies, although the concept is most clearly associated with **Jupyter**, which is a coding and presentation environment based on **Python**. Whatever the underlying programming environment, notebooks — or as **MdsX** uses the term, "interactive/digital notebooks" (**IDN**s) — have several software-engineering requirements, including a scripting environment and a data-visualization layer, wherein data sets or numeric models are transformed into **2D** or **3D** graphics (charts, diagrams, etc.). Moreover, the scripting layer needs to be connected to the data-visualization layer so that scripts can modify the data-to-graphics transformations. A further requirement is functionality to load pre-existing data sets from saved files or from a web resource.

Beyond these general features, **IDN** programming can take different forms and prioritize different styles of user interaction. The **MdsX** approach recognizes that it is often more convenient to interact with applications through **GUI** actions — buttons, tabs, context menus, and so forth — than by typing in commands (whether or not these are executed immediately in **REPL**, or "read-eval-print-loop", fashion, or are stored in scripts). As such, **IDN**s should not differ in design too noticeably from conventional **GUI** windows or dialog boxes — they should not be little more than

"REPLs with plots." On the other hand, rigorous GUI programming calls for a carefully organized set of mappings from potential user actions to application responses. Whether on the scripting level or the GUI coding, in short, implementations need a level of abstraction more general than the underlying event-handling and procedure-calling logic which forms the application's concrete operational behavior. This semi-abstracted layer can be described in terms of "meta-classes," "meta-objects," "tools," "transitions," "services," and so forth: the common denominator in different contexts is some notion of a structure which can be called a "meta-procedure," similar to an ordinary computational procedure in having inputs and outputs, but embodying a level of abstraction somewhat removed from concrete procedures. In particular, meta-procedures are not directly implemented; instead, some algorithm is necessary to determine, given a description of a meta-procedure with its outputs and context, what concrete procedure (or set of procedures) should actually run. Moreover, meta-procedures need some notion of delayed execution: there is a logical gap between "marking" (using the language of petri-net theory), i.e., fully specifying the input parameters consumed by a meta-procedure, and a meta-procedure's actual execution. As such, meta-procedural markings can be built up in stages, with input data coming from multiple sources (including scripts and GUI elements). For a concrete example, consider the process of filling out a web form, wherein entries typed in to the form fields are validated, one at a time, before the form can be submitted. In these cases, the step-by-step process of entering and validating individual fields corresponds to incremental marking, and "hitting the submit button" corresponds to meta-procedure execution.

In short — although different systems use different terminology — any IDN programming environment needs a mechanism to incrementally define and execute meta-procedure calls. The implementational foundations of that mechanism (hypergraphs, workflow engines, state monads, etc.) depend on the underlying programming environment. The MdsX approach borrows ideas primarily from HyperGraphDB and SeCo, which is a notebook-programming environment based on HyperGraphDB. As in SeCo, units of marking and execution are called *cells*. The main difference between MdsX and SeCo (apart from C++ instead of being the underlying programming language) is that MdsX cells are not intended, in the general case, to be typed in by programmers directly. Instead, MdsX cells are normally constructed behind the scenes, on the basis of GUI component state, user actions, or scripting input. However, once constructed, they can be manipulated like SeCo cells, both in terms of functionality and in terms of rationale: they can be used as a log of user actions, for undo/redo, for defining workflows, for generating scripts, and so on. In particular, the mappings from GUI actions to application handlers can be defined (and extended) by annotating the relevant GUI elements with meta-procedure cells. This also allows data sets to be annotated with micro-citations (which are discussed below).

As a C++ environment, MdsX uses an embedded "virtual machine" to interpret meta-procedure cells; application-level event handlers are not automatically exposed to a scripting interface as they would be in a or PYTHON environment. However, MdsX also supports scripting via a choice of languages, similar to SeCo. The primary scripting language used with MdsX is AngelScript,

although other C/C++ based languages (Embeddable Common Lisp, , etc.) can work as well. To support various scripting languages, modules loaded into MdsX need to provide a meta-procedural interface declaration, and the desired scripting language also needs a bridge to work with these declarations (which is generally usable across all datasets and modules). Such a bridge will be provided by default for AngelScript and (Embeddable Common Lisp), and similar tools could be implemented for other languages.

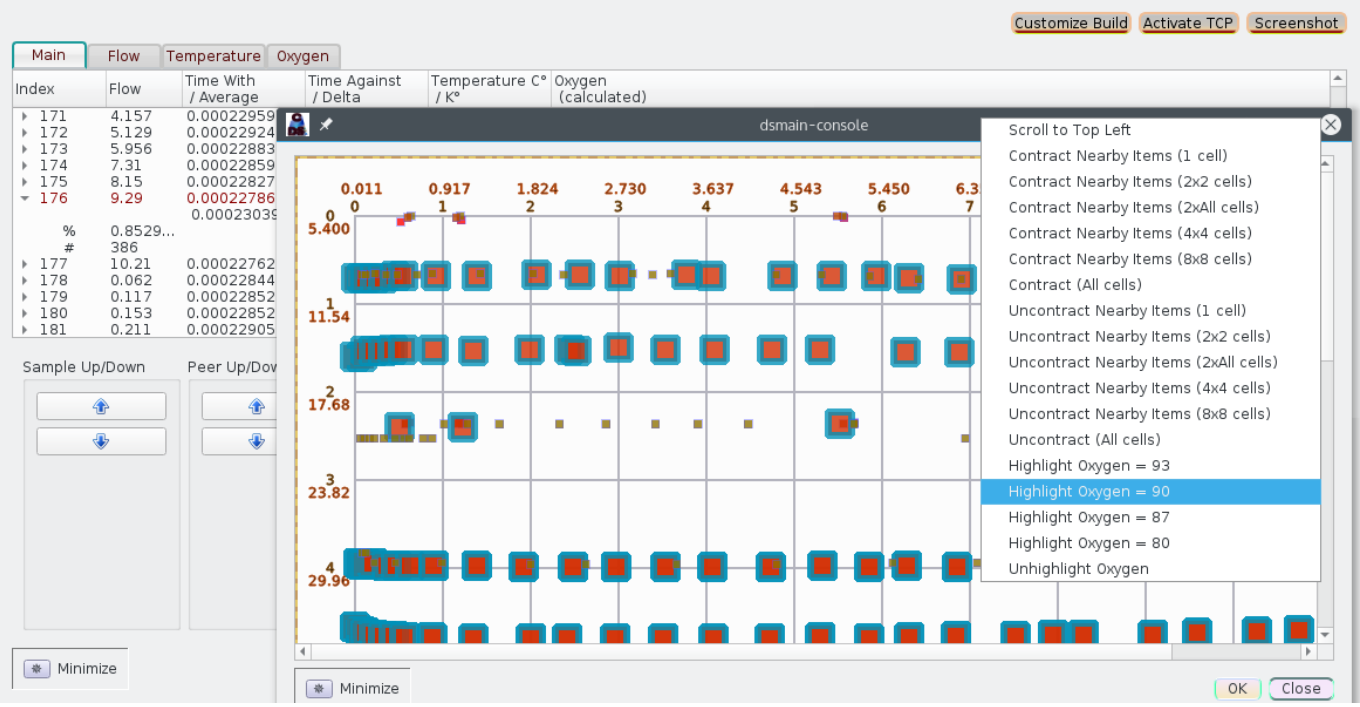
The typical MdsX notebook combines, at a minimum, some graphical element — such as an image to be analyzed and/or a plot/diagram to be populated with data — along with a user-interface "panel" for interacting with the graphics, and the overall application. This panel partially takes the place of a script-composition or REPL frame, although such a frame is implicitly present, normally behind the scenes (users can view it if desired). Notebooks can then load data files, and representations of the loaded data (e.g., text serializations) may then also become part of the notebook content, able to be visualized in their own frame. Notebooks in general then can have four varieties of frames (graphics views, interaction panels, data panels, and meta-procedure logs) although not every available frame may be explicitly constructed and/or visible at a given point in the user's session. There may also be multiple instances of graphics frames. In any case, the layout and state of these various frames — what frames are visible, and their current content — define notebook *state* which can be saved, restored, and shared. Loading a data set into a MdsX notebook therefore involves loading a particular initial state, defined as part of the data set, arranged in part to serve as a useful starting-point for users to explore and visualize the relevant data. Each of these kinds of frames corresponds to a particular aspect of software implementations, requiring its own strategies and paradigms. The following sections will review these various programming concerns one at a time.

Image Analysis and Data Visualization

The central graphical element of an MdsX notebook is either a 2D or 3D image loaded from an image file (in formats such as PNG, JPG, DICOM, TIFF, etc.), or else a 2D or 3D plot, chart, or diagram. The functionality of the notebook will therefore differ depending on whether the central graphics is an image loaded from a file (called an "image-based" notebook), or a data visualization constructed from a data set or some mathematical formulae (called a "diagram-based notebook"). A third option is an "object-based" notebook, whose central viewport contains a structured display of information, mostly in textual form (for instance, a table or tree view), but this section will focus on graphics-oriented notebooks (Figure 4 shows a contrast between an object-based view, in the background, and a graphics-based view, which has been opened floating above it).

Diagram-based MdsX notebooks can be implemented with different diagram-plotting engines; the default implementations support QT Charts (a built-in QT module) as well as the qtcustomplot

Figure 1: A graphics-based view (foreground) and object-based view (background)



and JKQCustomPlotter libaries. In short, diagram-based notebooks need to implement subclasses of **MdsX** frames for a navigation panel, graphics view, meta-procedure view, and data view, as well as a "**MdsX** diagram" subclass occupying the diagram view. In this case, the primary responsibility of the meta-procedural layer is to interface with functionality provided by the diagram/plotting engine. Since most coding details are derived from these engine's object models and classes, they lie mostly outside the scope of this paper.

Image-based notebooks, on the other hand, need to integrate several different areas of functionality. As such, setting up the image view is only one step in constructing such a notebook; additional programming is needed to support image annotation, analysis, and feature extraction. In order to integrate these different layers of functionality, **MdsX** provides a "Data Structure Protocol for Image-Analysis Networking" (**D-SPIN**) which defines communication rules between image-related software subsystems in Object-Oriented terms. **D-SPIN** objects are comprised of four more specific objects or layers, describing different aspects of the shared image and how it should be processed. These four layers are defined as follows:

Metadata Layer This object presents metadata describing the image format and acquisition. If the surrounding **D-SPIN** object represents an image series (rather than a single image), the metadata object should also declare the size of the collection and how individual images should be referenced. The metadata should include a file path or resource identifier asserting where the image can be acquired from (which in the case of a series can be a zipped folder or a list of resource paths). More specific metadata depends on the image or images' graphical format;

to properly load images in most formats (such as **PNG**, , **TIFF**, and **DICOM**) applications need to specify detail such as dimensions, resolution, and color depth. Of course, some of this information is stored internally within the image file (depending on its format), but certain formats require some metadata to be shared along with the image itself (moreover, it is often convenient to have basic information available without needing to extract it from binary image data). The details on which form of metadata are appropriate for which image format can be determined based on image-viewing code libraries, such as **libpng**, **libtiff**, or **DICOM** clients. If both end-points of a **D-SPIN** communication have the same libraries installed, the sending application will have a clear idea of how much supplemental data is needed over and above what will be read from image files directly. If there are uncertainties in library alignment between the two end-points, the sending application should consider serializing a more detailed summary of the image providing any information that would ordinarily be read from the image file.

Annotation Layer Almost all image analysis — whether done by humans or by softare — results in either some form of statistical representation of an image's properties, or a complex of data which presents information about (and may visually overlay) the image, particularly in the form of annotations. Image annotations are arrows, line segments, or **2D** closed shapes that call attention to some point or region inside the image, usually with some additional label or commentary. The basis of each annotation is therefore some zero-dimensional or two-dimensional region (or a set of zero-dimensional control points; or, occasionally, a one-dimensonsal line or curve), so annotations require a mechanism for designating regions. The same issues apply to asserting feature-vectors with respect to an image region rather than the image as a whole; accordingly, both annotations and feature vectors can be seen as equivalent varieties of constructions which isolate and then define data structures on zero-, one-, and/or two-dimensional subimages (feature vectors on the entire image can accordingly be treated as a special case).

Contextual Layer Contextual information associated with an image can include metadata or supplemental details that are not directly relevant to the image, but convey facts about how the image connects to a broader context where it was obtained, and for what purpose. An example of contextual data would be the part of **DICOM** headers that include patient or clinical information, rather than metadata about image format or dimensions.

Procedural Layer

In **MdsX**, **D-SPIN** objects are associated with image-based notebooks in that the notebook components (the navigation panel and graphics, data, and meta-procedural controllers) jointly refer to a common **D-SPIN** object for all image-related data. Image-analysis routines conducted within the notebook then may yield additional data structures bundled into the overarching **D-SPIN** object. This overarching object can then be exported or saved, alongside (and as an extension of) notebook

state.

Analytic operations available through an image-based notebook may be provided by the notebook itself, or by a host application where MdsX is embedded. In the latter case, the notebook needs to construct the proper calls to the host application, using the meta-procedural controller as a bridge to ambient capabilities. For instance, if an MdsX notebook is developed as a plugin to CAPTK, the notebook would interface with the host CAPTK application via the formats and programming constructs which CAPTK recognizes (specifically, the Common Workflow Language and the QT signal/slot mechanism). This specific scenario — embedding MdsX in CAPTK — is employed as a demonstration and case-study for embedding notebooks in host application in general. The CAPTK workflow protocol also forms a basis for the meta-procedural view and controllers, discussed next.

MdsX Meta-Procedure Controllers

The meta-procedural layer of an MdsX notebook is responsible for handling events generated by a corresponding navigational panel, or at least those events which have a non-trivial impact on notebook/session state and data. The visual representation of meta-procedural commands and history is provided by a meta-procedural "view," which is normally invisible, but notebooks may choose to allow users to "unhide" this view. The meta-procedural controller is responsible for generating the meta-procedural view (if applicable) and responding to user events within this view; it is also responsible for maintaining an inventory of objects summarizing available metaprocedures, GUI elements, and the mappings between them.

In general, the GUI elements in these meta-procedural mappings are referred to as "visual objects," and are represented in the meta-procedural controller context via application-unique identifiers (not raw pointers). Similarly, "meta-procedural objects" encapsulate information about meta-procedures themselves. This controller does not directly connect GUI events to event handlers; instead, it receives information about these connections when the notebook is loaded. The controller is however responsible for implementing *incremental execution* wrapping event callbacks (or any other relevant procedure). Incremental execution means that the controller may create temporary "execution contexts" and incrementally build up the data which, given a sufficiently complete "marking," can lead to the meta-procedure being "fired." Each preliminary stage — that is, each pre-firing addition to the execution context — may in turn be generated by events originating elsewhere in the application (canonically, the navigation panel), and the meta-procedural controller should model both the history of these pre-firing stages and the origin and nature of the events which triggered them. An execution context may then be *reified*, representing the cumulative pre-firing stages as a data structure that can be matched to a meta-procedure's outcomes: for example, noting the inputs or steps producing the specific appearance of a diagram or image in the graphics view, or the parameters configured to instantiate a workflow model. Reified meta-procedural

execution contexts can then be shared as objects with components responsible for sharing or preserving information about the notebook. For instance, a notebook graphic may be included in a publication; the reified context could then be associated with that image as an annotation, and used to reconstruct notebook state if the notebook is launched from a document viewer in the context of the published graphic.

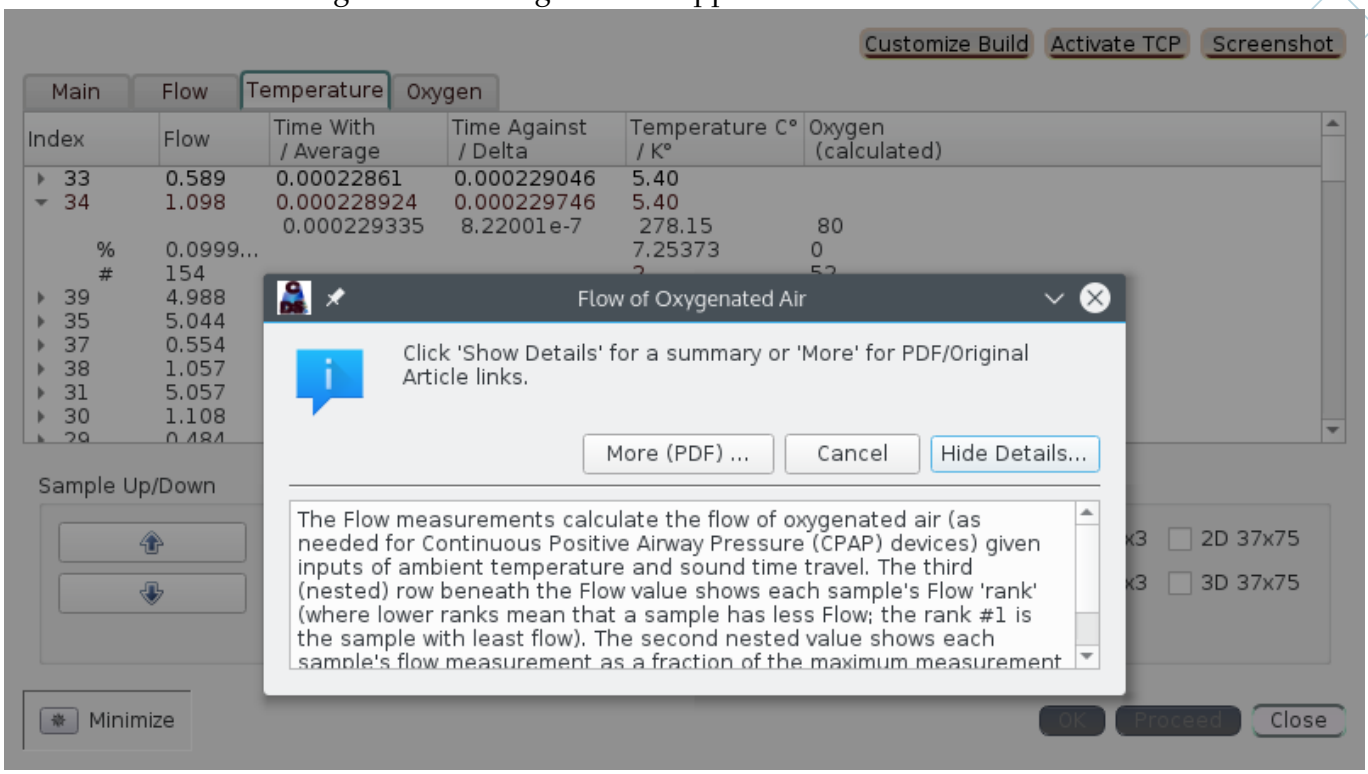
In general, the information represented by the meta-procedural controller is not only relevant for the reactive operations of the notebook, responding to user actions; it also serves to document the notebook's properties, and potentially to connect the notebook with data sets and/or publications. Many operations which can be performed within a notebook are associated with a given scientific or theoretical concept, or a statistical parameter modeled within a data set. As such, it is possible for the notebook to maintain a list of these concepts, so as to create an interactive glossary or to interoperate with a document viewer. For instance, Figure 4 shows a context-menu action based on the concept of "oxygenated air flow," which is also discussed in the scientific article on which the depicted data set is based. This concept also has a visual expression in one table column shown (in the background) on Figure 4. As demonstrated in Figure 5, the data-set application includes code to explain technical concepts in pop-up dialog boxes, and also to link to the page/paragraph in the article where that corresponding concept is first (or most thoroughly) defined/mentioned. Establishing these conceptual connections between an **MdsX** notebook, data set, and technical publication is facilitated by annotating both meta-procedural capabilities and **GUI** elements with references to relevant technical/scientific concepts; these annotations, when defined, are represented through the meta-procedural controller.

DigammaDB and the HGXF Format

CR2 will introduce a new database engine for preparing the information provided within the repository (called DigammaDB, or ςDB for short) as well as the Hypergraph Exchange Format (**HGXF**). In general, **HGXF** files can be generated from ςDB instances, capturing the state of the database at a moment in time. ςDB could therefore be used to store research data. When scientists choose to publish data, they would then output the information from their database into **HGXF**, using the resulting files as the published raw data. In the **CR2** context, ςDB will be used to merge data from multiple files into a single database, yielding **HGXF** output forming most of the raw data republished in **CR2**.

Operationally, ςDB is designed to emulate the programming interface provided by several existing databases and hypergraph libraries — for instance, HyperGraphDB ([?]), WhiteDB ([?]), and HgLib ([?, p. 9]). That is, ςDB is designed so that existing code using these technologies can be adopted for ςDB with relatively little effort. ςDB also introduces some new concepts and structuring

Figure 2: Linking Dataset Applications to Publications



features, which will reviewed below. In addition to prior technologies such as HyperGraphDB, ςDB draws on theoretical work connected to hypergraphs and their value as multi-paradigm, general-purpose metamodels. The reach of hypergraph theory encompasses several paradigms for modeling scientific data, such as Conceptual Graph Semantics (see [?]) and Conceptual Space Theory (which is linked to hypergraphs in work summarized by [?], where Conceptual Space semantics is paired with hypergraph categorial grammar). ςDB therefore introduces modeling elements designed to capture scientific details (such as dimensional analysis and units of measurement). When discussing graph structures and programming techniques, ςDB draws terminology from these scientific perspectives as well as from existing Hypergraph database engines.

5.1 Scientific Features of DigammaDB

ςDB has no specific connection to SARS-COV-2 or Covid-19; it is conceived as a general-purpose database engine that can facilitate application development across many domains and industries. However, ςDB is designed with exceptional attention to scientific research and software, with respect to metadata, GUI integration, its representation of files and file-types, and interoperability with technologies related to publishing and open-access research data. A full enumeration of ςDB's scientific focus is outside the scope of this outline, but certain features are specifically relevant to CR2, so they can be discussed here:

From-the-Ground-Up QT Integration All GDB classes natively interoperate with QT (a leading cross-platform application-development and GUI framework). Programmers then have access to features like QDataStream and QT meta-objects for binary serialization of C++ objects for persistence in the database. Robust QT support also makes it easy to design GUI classes for interacting with GDB data, and for employing GDB as a technology for managing and storing application state. This is important in the scientific computing context because most scientific software is designed as native desktop-style applications needing special-purpose GUI classes (in this environment it is usually not possible to adopt techniques such as HTML page templates which are commonplace in the web-programming context for creating user views onto database objects). Therefore, implementing custom GUI logic is an important part of the development requirements for scientific applications, and it is helpful to interface with a database engine prioritizing that aspect of software engineering.

Projections and Scientific Annotations GDB adopts the HyperGraphDB notion of *projections*, or descriptions of individual fields within a complex object that can automate the extrapolation of binary-serialization algorithms (which in turn allows complex objects to be stored in GDB alongside primitive values like strings and numbers). GDB also uses projections as a basis for introducing metadata associated with Conceptual Spaces and Conceptual Space Markup Language (CSML) as mentioned earlier. In particular, projections can be annotated with statistical/quantitative details such as dimensions (e.g., base quantities and units of measurement), scales or "measurement levels" (in CSML terms, particularly the distinction between nominal, ordinal, interval, and ratio axes), minima and maxima, probability distributions, and the aggregation of isolated units into complex dimensions, or CSML "domains" (vectors, tensors, area/volume spans, and so forth).

Data Microcitations Most scientific data is formally or informally linked to peer-reviewed publications which present research findings. Published data sets make these connections explicit, by joining document identifiers and URLs with the corresponding identifiers for designating and locating data sets. These connections are however coarse-grained, applying only to documents and data sets in their entirety. By contrast, an emerging trend in publishing is to link portions of publications — such as individual sentences, paragraphs, or figure illustrations — with smaller parts of a data set (which could be an individual record/sample, or a table column, or some conceptually related group of records or columns). To support microcitations, there must be some formal mechanism to individuate small parts of a data set. This structure is provided internally by GDB: there are multiple microcitable "zones" in a database, which introduce encodings for microcitation targets that become exposed to publications via HGXF. Here "zones" refer to aspects or portions of a database that may be cited individually, apart from the database as a whole: records/atoms, types, projections, subgraphs, software components implementing a digamma-application interface (analogous to HGApplication in HyperGraphDB), among others. These entities can be given identifiers which are unique not only in the context of a single database, but if needed over a larger corpus (e.g. an archive of research data) so that the relevant atoms/types/projections can

be referenced from publication texts (and similar resources). Such references can also be linked to GUIs; one can for instance say that the data currently visible in a GUI window or dialog box is a view onto a specific database atom, information that could be used for debugging, storing application state, inter-application networking (e.g. linking PDF viewers for research papers with applications to view research data), and so forth.

Memory and Persistence Models ÇDB can be run in several different "modes", which determine how data is stored in memory while an application is running and/or is stored in files. If desired, ÇDB can be used as an "in-memory" database which does not maintain persistent file state at all. This feature can be useful in a research-data context where all information derives from data files. In this scenario, a ÇDB instance can be loaded from parsed serializations (e.g., HGXF files) without using other file-system resources. The database engine can then be adopted as a tool for accessing and manipulating the "infoset" derived from these data files, analogous to XML libraries interfacing with a Document Object Model.

Scientific-Computing Interface Description Following HyperGraphDB, ÇDB establishes a "type system" unique to each database, which is responsible for translating application-level types to database atoms. The type system used by ÇDB is actually more detailed (compared to HyperGraphDB) and allows for the description of procedure signatures and the declaration of conceptually related procedural groupings, which collectively enable interface definition as persistent data within the database itself. ÇDB likewise supports a notion of "meta-procedures", which are indirectly derived from Alexandru Telea's *metaclasses* and *dataflow interfaces* (see [?]). Moreover, ÇDB introduces a notion of "channels", which are a higher-level graph structuring feature (see [?, Chapter 3]) that may, in particular, be applied for the semantic annotation of function signatures so as to refine interface documentations. Collectively, these features enable each ÇDB database to store information about procedures used to access and interact with their stored information, to facilitate the implementation of scientific workflows and GUIs for accessing and using ÇDB data.

5.2 Programming with HGXF Files

The prior outline described some of the principle features of ÇDB, which in turn influenced the design of HGXF, insofar as one goal of HGXF is to serialize ÇDB instances. However, HGXF files are also generic resources for serializing research data (or any other technical information), and ÇDB includes libraries for using HGXF (not necessarily in the context of ÇDB instances). Therefore, HGXF deserves a brief overview in its own right.

Like any hypergraph meta-model, HGXF represents information on two levels. On the one hand, each "node" is (in genral) a synthesis of multiple smaller parts. In HGXF, the parts of a hypernode are *structure fields* and *array fields*, where the former are roughly analogous to named columns, and the latter are usually either units within expandable collections (such as lists, queues,

and dictionaries) or fixed-size arrays of numeric fields with similar dimensional qualities (e.g., vectors representing point in space). **HGXF** files define "types", which are internal to the file (but may be associated with GDB types) and describe the layout of fields within the containing hypernode (in particular, the number and names of structure fields, and restrictions on the number of array fields, if any). **HGXF** types may also provide hints about how to best encode the parsed data in running memory.

The structure of **HGXF** files is specified by a Reference Implementation using **C++** to parse and manipulate **HGXF** data. In general, this code library mixes functional and object-oriented programming styles. The central procedures in this library are ones to isolate single hypernodes, by graph traversal or queries, and then procedures to operate on array and structure fields contained within hypernodes. The basic interface for this latter context involves passing criteria for selecting one or more fields, along with a callback function which will be called with the vector of matching fields (or else a procedure to operate on fields one at a time). In this functional style, **HGXF** hypernodes can be seen as monads encapsulating access to their internal structure and array fields (or "hyponodes").

On a higher level, applying not to the scope of a single hypernode but to groups of hypernodes, **HGXF** introduces several higher-level structuring elements (mostly derived from corresponding GDB features). In particular, *channels* are aggregates of edges, and *frames* are contexts for defining edges and nodes. In GDB, every node and edge is constructed in the context of a frame, which *may* (but need not) offer either additional semantic detailing or callback functionality to add code affecting how graph data is constructed. All graphs have a "default" frame that implies no special semantic context (i.e., all edges are asserted relations deemed to apply in general, with no contextual filter) and that utilizes general-purpose algorithms for edge and node construction. Developers can introduce more specialized frames as desired, and construct nodes and edges in these tailored contexts. Such frames can then be identified in **HGXF** as the context where a given node or edge serialization applies. Hypernodes may also be annotated within frames to identify a conceptual, operational, or semantic role of a node within some larger node-collection. Indeed, GDB introduces the concept of a *virtual frame* to capture logical patterns implicit in certain node-annotations which are not explicitly designated by frame objects allocated within the database.

The default behavior for edge-construction (which can be overridden within individual frames) conceives edges as **RDF**-style "triples", where a directed pair of nodes is annotated with a simple label (which may or may not be defined within a controlled vocabulary) and/or a more complex object; edges can be marked with hypernodes of their own. The source and target hypernodes of a hyperedge correspond to the source and target node-sets defined in most mathematical treatments of hypergraph theory. In libraries such as HgLib, which (compared to graph/hypergraph databases) are more directly based on this mathematical theory, hyperedges are defined by providing one or two (for undirected or directed edges respectively) sets of nodes. GDB supports this construction

as well, essentially forming hypernodes "on the fly", but in this case the fields within these auto-constructed hypernodes are "proxies" for other hypernodes (possibly three or more) linked by the edge. Proxies can also be defined for more complex objects, such as frames and subgraphs, allowing multi-dimensional, nested graph structures if these are desired for a particular domain model.

HGXF also supports microcitations, which are discussed above. By designating parts of a data set as microcitation targets, HGXF can be integrated with annotation systems designed for scientific publications. These features allow document annotations — which are used, for instance, to connect text in a research paper with scientific concepts and Ontologies — to connect also with microcitable elements within data sets and Research Objects which serialize data via HGXF. In this use-case, HGXF works in conjunction with a related protocol, the Annotation Exchange Format (AXF), which is also initially developed to support the CR2 implementation. In CR2, AXF will be used to connect Covid-19 research data with publications where the corresponding data sets are introduced to the scientific community. The AXF format and design principles will be discussed further in the next section.

The AXF Platform

The AXF Platform (hereafter called AXF) is a toolkit for hosting full-text, open access publications, with an emphasis on scientific, academic, and technical documents. At the core of an AXF Publication Repository is a collection of files in a machine-readable AXF Document Format (AXFD), which are paired with human-readable PDF documents as well as supplemental multi-media and metadata files. Depending on institutional requirements, an AXF repository may be the primary storage resource for the contained publications, or an adjunct resource whose documents are linked to publications hosted elsewhere. In the second scenario, the primary goal of an AXF repository is to host manuscripts in AXFD format, along with software to aid viewing and text-mining of the associated publications.

AXFD therefore has two distinct purposes: (1) to aid in text and data mining (TDM) of full publication text (along with research data that may be linked to publications), and (2) to enhance the reader experience, given e-Reader software (canonically, PDF viewers) which are programmed to consume AXF information. To (1) aid in text mining, AXFD documents can be compiled into different structured representations, yielding document versions that can be registered on services such as CrossREF TDM and SemanticScholar. Given a Document Object Identifier, text-mining tools can therefore readily obtain a highly structured, machine readable version of the publication, which may then be used as the basis for further text-mining and NLP operations. Simultaneously, to (2) improve reader experience, the AXF platform generates numeric data linking semantically significant text locations to PDF viewport coordinates (such text locations include annotation, quotation, or citation start/end points and paragraph or sentence boundaries — collectively dubbed a Semantic Document

Infoset, or SDI). This SDI+Viewport (SDIV) information can then be used by PDF applications to provide contexts for word searches, to localize context menus, to activate multi-media features at different points in the text, and in general to make PDF files more interactive. Data sets composed with the aid of AXF tools may include source code for a PDF viewer (an extension to XPDF) capable of leveraging AXF data.

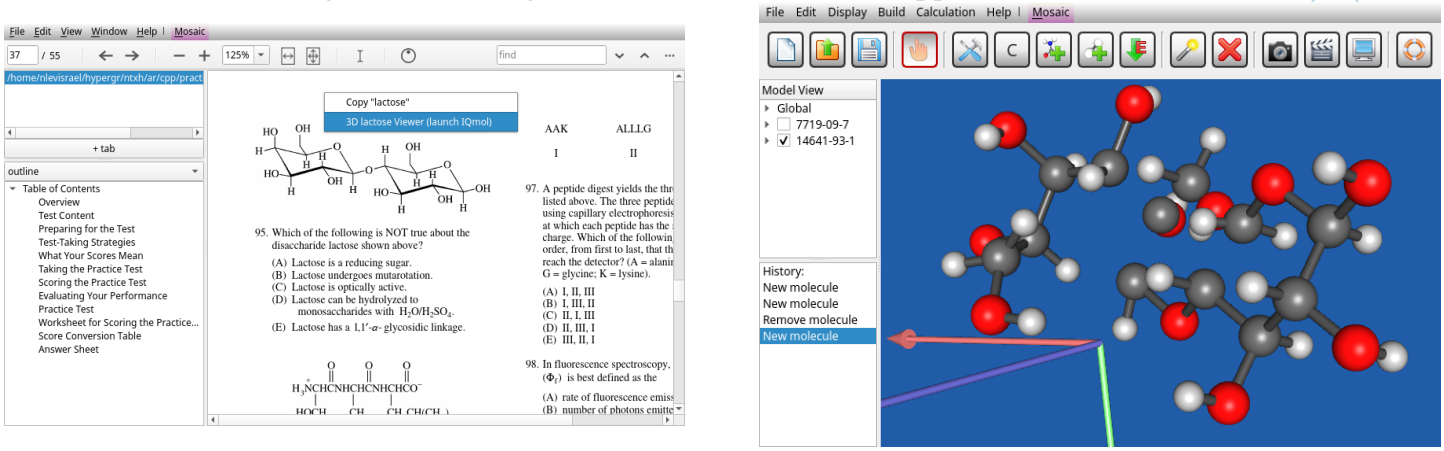
In addition to the AXFD document format, the AXF platform includes the Annotation Exchange Format itself, a protocol for defining and sharing annotations on full-text publications. AXF differs from other annotation-representation strategies by (1) providing more detail concerning the location of annotated text segments, in the surrounding publication context, and (2) supplying annotation data in multimedia or microcitation formats which extend beyond conventional "controlled vocabularies". In terms of (1) publication context — that is, the annotation *target* (using terminology from the Linguistic Annotation Framework, or LAF) — AXF represents SDIV information as introduced above; this data supplements the node/index coordinates used by traditional annotation mechanisms. With respect to (2) annotation metadata — or the annotation *body* (again using LAF terminology) — AXF introduces models for multimedia assets, software components, and data set content, which may be linked to annotation targets. With this additional metadata, annotations may be used in application-development environments, not only for text mining.

The following sections will (1) outline AXF and AXFD in greater detail, (2) describe how AXF repositories can unify publications sharing similar themes, scholarly disciplines, or coding requirements, and (3) describe features for data-set publication in the context of AXF repositories.

6.1 AXF Documents

The AXFD format for describing document content and structure is designed to be a "Pivot Representation" in the sense of LAF (see [?]). In particular, AXFD can represent the structure of both XML (including several XML flavors used in publishing) and LaTeX. Technically, AXFD does not prescribe any specific input format; instead, a document is considered an instance of AXFD if it can be compiled into a Document Object satisfying interface requirements. A C++ reference implementation anchors the AXF Document Object Model; nodes in this implementation have facets combining LAF, XML, and LaTeX. In practice, AXFD manuscripts are then converted via LaTeX to PDF, and simultaneously compiled to XML representations so as to generate machine-readable, structured full-text versions of the manuscripts. Authors can choose to compose AXFD papers to conform with several common publication XML standards, such as JATS (Journal Article Tag Suite), SciXML [?], and IeXML [?] (the latter is an annotation-oriented XML language used by the BeCAS project [?]). Authors or editors may further define or model scientific concepts via strategies focused on computer simulation (in the broad sense as defined, for instance, in [?]) or statistics (e.g., Predictive Model Markup Language or Attribute-Relation File Format), and/or conceptual semantics, such as Conceptual Space Markup Language (which was inspired by the linguist Peter Gärdenfors, but developed

Figure 3: Linking PDF Files with Scientific Applications



in a scientific/mathematical framework in e.g. [?], [?], [?], and [?]).

One distinct feature of **AXF** is that LaTeX and **XML** generation are chained in a pipeline: the LaTeX and subsequent **PDF** generation steps yield auxiliary data, which includes **PDF** viewport data, that can be subsequently incorporated into **XML** views onto the documents. Specifically, **AXFD**-generated LaTeX files include notations for semantic annotations and for sentence boundaries, implemented via LaTeX commands which, as one processing step, write **PDF** coordinates to auxiliary files. The resulting data is then read by a **C++** program which collates annotations and sentence-boundaries into a vector of data structures indexed by **PDF** page numbers, creating a distinct file for each page, and zips those files into an archive which can be distributed alongside (or embedded inside) the **PDF** publications. Simultaneously, sentences, paragraphs, annotations, and other semantically significant content (such as quotations and citations) are assigned unique ids and compiled into their own data structures (from which machine-readable **XML** full-text may be generated). These **XML** files may then be hosted and/or registered on **TDM**-oriented services such as CrossRef. At the same time, unique identifiers unify this **XML** data (focused on text mining) with **PDF** viewport data (focused on reader experience). The goal of such integration is to incorporate text-mining results so as to enhance reader experience. For example, Named Entity Recognizers might flag a word-sequence as matching a concept within a controlled vocabulary. Via the relevant paper's **SDI** model, this annotation may be placed in a proper semantic context — for example, obtaining the text of the sentence where the Named Entity occurs. This semantic information may then be used by a **PDF** viewer — e.g., providing a context menu option to select the sentence text, when the context menu is activated within the rectangular coordinates of the annotation itself.

As a representation of annotation data structures, **AXF** ensures that **SDI** and viewport data is included among annotations wherever this data is available. This facilitates the integration between text-mining tools and **PDF** viewer software, which in turn enhances reader experience. As mentioned earlier, every annotation can be placed in a semantic context (e.g., the text of the surrounding sentence), which provides useful reader features such as one-click copying of sentences to the

clipboard. Other reader-experience enhancements involve multimedia assets. As a concrete example, suppose a paper includes mention of a chemical; that particular keyword can accordingly be flagged for annotation. As one encoding of the corresponding scientific concept, the annotation can include the chemical's Chemical Abstract Service Reference Number, via which it is possible to obtain Protein Data Bank (PDB) files to view the relevant molecular structure in 3D. In sum, annotations supply a constellation of data — in this example, concepts may be linked not only to identifiers in cheminformatic ontologies, but also to CAS reference numbers and thereby to 3D graphics files — which facilitate interactive User Experience at the application level, not only document classification at the corpus level. Once a chemical compound (mentioned in a publication) is linked to a PDB file (or any other 3D format) the PDF viewer may include options to for the reader to connect to software or web applications where the corresponding visuals can be rendered. Via AXF, the relevant document-to-software connections are asserted not only on the overall document level, but on the granular scale of the precise character and PDF viewport coordinates where the relevant annotation is grounded (Figure 3 illustrates such capabilities in the context of a chemistry publication — specifically, test-preparation materials for the Chemistry GRE exam).

To support this kind of multimedia functionality, AXF standardizes a Plugin Framework, dubbed "MOSAIC", allowing programmers to embed code which can parse and respond to AXF annotations in different scientific and document-viewer applications. MOSAIC allows different applications to inter-operate; in particular, PDF viewers can share data with scientific applications that can render files in domain-specific formats such as PDB. This application networking protocol is considered part of the AXF annotation model, because application-oriented information is computationally relevant for many concepts encountered in scientific and technical environments. For instance, one aspect of cheminformatic data is that many chemical compounds are modeled by PDB, MOL, or CHEMXML files, which in turn are associated with software applications that can load those file types. Such inter-application networking data is relevant to PDF viewers when they display manuscripts with annotations that suggest links to special file types and their applications; the viewers can employ this information to launch and/or communicate with the corresponding software. AXF is designed to facilitate implementation of application-networking protocols as an operational continuation of processes related to obtaining and consuming annotation data.

The AXF document model, at the manuscript-structure level, is paired with a novel "Hypergraph Text Encoding Protocol" (HTXN) operating at the character-encoding level. Within the HTXN protocol, an annotation target is a character-index interval in the context of an HTXN character stream. On that basis, HTXN treats documents as graphs whose nodes are ranges in a character stream, where text can be recovered as an operation on one or more nodes (e.g., the text of a sentence is derived from a pair of nodes representing the sentence's start and end). HTXN code-points are distinguished in terms of their semantic role, which may be more granular than their visible appearance — for example, a period glyph is assigned different code-points depending on whether it marks a sentence-ending punctuation, an abbreviation, a decimal point, or part of an ellipsis. Procedures are then

implemented to represent text in different formats, such as ASCII, Unicode, XML, or LaTeX. In contrast to a format such as Web Annotations, any particular human-readable text presentation (including ASCII) is considered a *derived* property of the annotation, not a foundational representation.

AXFD manuscripts do not need to utilize HTXN for character data, but HTXN simplifies certain AXF operations, such as identifying sentence boundaries. In particular, HTXN provides distinct code-points for end-of-sentence punctuation, so that sentence-boundary detection reduces to a trivial search for those particular code-points. Proper HTXN encoding requires that authors follow certain simple heuristics — e.g., that end-of-sentence periods should be followed by two spaces and/or a newline, whereas other uses of a period character should precede at most one space. Aside from the goal of preparing documents for text-mining machine-readability, such conventions are appropriate even for basic typesetting, because non-punctuation characters have their own kerning rules (this is why LaTeX provides a distinct command for non-punctuation glyphs that would otherwise be read as punctuation characters). HTXN hides these typesetting details within its character-encoding schema, which is then useful both for producing professional-caliber LaTeX output and for identifying SDI details (such as sentence boundaries) which with less rigorously structured text would need elaborate text-mining or NLP algorithms.

Each AXFD document is, in sum, associated with an aggregate of character-encoding, annotation, document-structure, and PDF viewport information. The AXF platform uses code libraries to pull this information together as a runtime object system, so that any application which loads an AXFD manuscript can execute queries against the corresponding collection of AXF objects (queries such as obtaining the sentence text around an annotation, obtaining the concave-octagonal viewport coordinates for a sentence,[1] obtaining application-networking information for an annotation, etc.) In addition to such runtime data, AXF platforms can compile the full suite of information into machine-readable files for text and data mining. These files, collected across a corpus of multiple documents, then form the backbone of an AXF publication repository, as will be discussed next.

6.2 AXF Publication Repositories

The AXF platform is designed for hosting collections of publications sharing a common academic or technical focus. When AXF is used in the context of a general-purpose text and/or data repository, the AXF platform is designed to work with collections that are organized into separate projects or topics, each giving rise to an archive or corpus of publications. Insofar as these corpora internally share a common theme or focus, they can be associated with their own ontologies, code libraries, annotation

[1] In the general case, sentence coordinates are concave octagons because they incorporate the line height of their start and end lines; in the general case sentences share start and end lines with other sentences, while also including whole lines vertically positioned between these extrema. A sentence octagon roughly corresponds with the screen area where a mouse/pointer action should be understood as occurring in the context of that sentence from the user's point of view — implying that the user would benefit from context menu options pertaining specifically to that sentence, such as copy-to-clipboard.

models, and application-networking protocols, based on the sorts of applications and data structures commonly used in the corresponding scholarly discipline. In some cases, publishers may choose to package an entire archive of research papers (perhaps along with research data) as a single downloadable resource. **AXF** allows publishers to construct such Research Archives following the structure of existing examples such as the **ACL** (Association for Computational Linguistics) Anthology or the recent **CORD-19** corpus. This latter archive is a useful case-study in both the possibilities and limitations of existing publication-repository technology, so it is reviewed here in more detail.

CORD-19, curated by the Allen Institute for Artificial Intelligence, was spearheaded by a White House initiative to centralize scientific research related to **COVID-19** (see [?]). The collection was formulated with the explicit goal of promoting both *text mining* and *data mining* solutions to advance coronavirus research, so that **CORD-19** is intended to be used both as a document archive for text mining and as a repository for finding and obtaining coronavirus data for subsequent research. Although novel research is being incorporated into **CORD-19**, many of the articles reproduced in this corpus are older publications related to coronaviruses and to SARS in general, not just to the current pandemic. As a result, the full-text versions of these publications were retroactively aggregated into a single archive due to the unanticipated emergence of a coronavirus crisis, with the full text often obtained from **PDF** files rather than from structured representations (such as **JATS**) explicitly intended for text mining.

This archival methodology results in **CORD-19** being limited as a **TDM** framework. These limitations include the following:

Transcription Errors Transcription errors can easily result from trying to read scientific data and notations based on **PDF** files — or on full-text representations using relatively unstructured formats such as **XOCS** (the response-encoding format for the ScienceDirect **API**). Transcription errors cause the machine-readable text archive to misrepresent the structure and content of documents. For instance, there are cases in **CORD-19** of scientific notation and terminology being improperly encoded. As a concrete example, "**2'-C-ethynyl**" is encoded incorrectly in one **CORD-19** file as "**2 0 -C-ethynyl**" (see [?] for the human-readable publication where this error is observed; the corresponding index in the corpus is 9555f44156bc5f2c6ac191dda2fb651501a7bd7b.json). To help address these sorts of errors — which could stymie text searches against the **CORD-19** corpus — it is obviously preferable to archive structured, machine-readable versions of publications, using a platform such as **AXF**.

Converting Between Data Formats Although the **CORD-19** corpus is published as **JSON** files, many text-mining tools such as those reviewed in [?] recognize inputs or produce outputs in alternative formats, such as **XML**, **BIOC**, **CONLL** (Conference on Natural Language Learning), or **JSON** trees with different schema than **CORD-19**. For this reason, rather than providing data with one single representational format, it is better to encode the data along with code libraries that can

express the data in different formats as needed for different TDM ecosystems.

Inconsistent Annotations The structure of CORD-19 allows text segments to be defined via a combination of JSON file names, paragraph ids, and character indices. This indexing schema is used for representing certain internal details of individual articles, such as citations, but is not explicitly defined as an annotation target structure for standoff annotations against the archive as a whole. This problem could also be rectified with code libraries that map index targets to file handles and character pointers.

Limited Support for Research Data-Mining Even though many papers in CORD-19 are paired with published data sets, there is currently no tool for locating research *data* through CORD-19. For example, the collection of manuscripts available through the Springer Nature portal linked from CORD-19 includes over 30 COVID-19 data sets, but researchers can only discover that these data sets exist by looking for a "supplemental materials" or a "data availability" addendum near the end of each article. These Springer Nature data sets encompass a wide array of file types and formats, including FASTA (which stands for Fast-All, a genomics format), SRA (Sequence Read Archive, for DNA sequencing), PDB (Protein Data Bank, representing the 3D geometry of protein molecules), MAP (Electron Microscopy Map), EPS (Embedded Postscript), CSV (comma-separated values), and tables represented in Microsoft Word and Excel formats. To make this data more readily accessible in the context of CORD-19, it would be appropriate to (1) maintain an index of data sets linked to CORD-19 articles and (2) merge these resources into a common representation (such as XML) wherever possible. This research-data curation can then be treated as a supplement to text-mining operations. In particular, queries against the full-text publications could be evaluated *also* as queries against the relevant set collection of research data sets.

Wrappers for Network Requests Scientific use of CORD-19 will often require communicating with remote servers. For example, genomics information in the COVID-19 data sets (such as those mentioned above that are available through Springer Nature) is generally provided in the form of accession numbers which are used to query online genomics services. Similarly, text mining algorithms often rely on dedicated servers to perform Natural Language Processing; these services might take requests in BIOC format and respond with CONLL data. As another case study epidemiological studies of COVID-19 may need to access APIs or data sets such as the John Hopkins University "dashboard" (see https://coronavirus.jhu.edu/map.html, which is paired with a GIT archive updated almost daily). To reduce the amount of "biolerplate code" which developers need for these networking requirements, an archive's text-mining code could provide a unified framework with which to construct web-API queries, one that could be used across disparate scientific disciplines (genomics, NLP, epidemiology, and so forth).

Many of these limitations observed in CORD-19 reflect the fact that this corpus was prepared

as raw (text) data, without any supporting code. By contrast, recent initiatives — such as the Research Object protocol (see [?]) and FAIR ("Findable, Accessible, Interoperable, Reusable"; see [?]) — encourage authors to publish code and data together, so that the computing environment needed to process published data is provided within the data set itself. The CORD-19 limitations accordingly provide an example of why Research Objects, rather than raw data sets, should be preferred for data publication in the future. The Research Object model is usually defined in the context of a single publication, but the paradigm applies equally well to corpora encompassing many single articles. That is, AXF is structured so that Research Archives can be designed as higher-scale Research Objests, wherein the document collection is bundled with supporting code and an overall computing and software-development environment. Such archive-specific SDKs would include AXF-specific code as well as libraries or applications often utilized in the academic disciplines relevant to the archival subject areas. The AXF platform especially promotes the design of domain-specific SDK which are *standalone* and *self-contained*, with minimal external dependencies. As much as possible, users should not have to install external software to utilize data provided along with an AXF repository; instead, the needed data-management tools should be provided in source-code form within the archive itself.

Each AXF repository, then, should bundle numerous applications used for database storage, data visualization, and scripting. The goal of this application package would be to provide researchers with a self-contained computing platform optimized for scientific research and findings related to the archived publications. Archival SDKs should try to eliminate almost all scenarios where programmers would need to perform a "system install"; for the most part, the entire computing platform (including scripting and database capabilities) should be compiled from source "out-of-the-box". While the actual libraries and applications bundled with an archive would depend on its topical focus, the following is an example of components that would be appropriate in many different SDK:

- XPDF: A PDF viewer for reading full-text articles (augmented with CORD-19 features, such as integration with biomedical ontologies);
- QT: The QT library is a cross-platform Application-Development framework and GUI toolkit commonly used for scientific applications (XPDF is one example of a QT-based document viewer). Almost any data set can be accompanied with QT code for data visualization, so that readers would not have to install additional software. For its part, QT can be freely obtained and, once downloaded, resides wholly in its own folder (there is no install step which modifies the user's system); as such, QT along with individual archive SDKs function as standalone packages, although optimally the SDKs would be updated along with new QT versions.
- AngelScript: An embeddable scripting engine that could be used for analytic processing of data generated by text and data mining operations on CORD-19 (see [?]);
- WhiteDB: A persistent database engine that supports both relational and NoSQL-style archi-

tectures (see above);

- MeshLab: A general-purpose 3D graphics viewer;
- LaTeXML: a LaTeX-to-XML converter;
- PositLib: a library for use in high-precision computations based on the "Universal Number" format, which is more accurate than traditional floating-point encoding in some scientific contexts (see [?]).

To this list one might add components specific to various scientific fields: IQMOL for chemistry and molecular biology, for example, or open-source libraries such as EpiFire or Simpact (for Epidemiology), UDPIPE (for CONLL), and so forth. Here again the priority would be for self-contained components with few external dependencies — particularly libraries programmed in C or C++, which are the languages best positioned to be a common denominator across diverse research projects (of course, many scientific C++ libraries have wrappers for languages like R or Python that researchers may be more comfortable using). In general, Research Archive code should be (1) *self-contained* (with few or no external dependencies, as emphasized above); (2) *transparent* (meaning that all computing operations should be implemented by source code within the bundle that can be examined as code files and within a debugging session); and (3) interactive (meaning that the bundle does not only include raw data but also software to interactively view and manipulate this data). Research Archives which embrace these priorities attempt to provide data visualization, persistence, and analysis through GUI, database, and scripting engines that can be embedded as source code in the archive itself.

It is worth noting that a data-mining platform requires *machine-readable* open-access research data (which is a more stringent requirement than simply pairing publications with data that can only be understood by domain-specific software). For example, radiological imaging can be a source of COVID-19 data insofar as patterns of lung scarring, such as "ground-glass opacity," are a leading indicator of the disease. Consequently, diagnostic images of COVID-19 patients are a relevant kind of content for inclusion in a COVID-19 data set (see [?] as a case-study). However, diagnostic images are not in themselves "machine readable." When medical imaging is used in a quantitative context (e.g., applying Machine Learning for diagnostic pathology), it is necessary to perform Image Analysis to convert the raw data — in this case, radiological graphics — into quantitative aggregates. For instance, by using image segmentation to demarcate geometric boundaries one is able to define diagnostically relevant features (such as opacity) represented as a scalar field over the segments. In short, even after research data is openly published, it may be necessary to perform additional analysis on the data for it to be a full-fledged component of a machine-readable information space.[2] To deal with this sort of situation, AXF equips SDKs with a *procedural data-modeling vocabulary* that

2 This does not mean that diagnostic images (or other graphical data) should not be placed in a data set; only that computational reuse of such data will usually involve certain numeric processing, such as image segmentation. Insofar as this subsequent analysis is performed, the resulting data should wherever possible be added to the underlying image data as a supplement to the data set.

would both identify the interrelationships between data representations and define the workflows needed to convert research data into machine-readable data sets.

Another concern in developing an integrated Research Arcive data collection is that of indexing documents and research findings for both text mining *and* data mining. In particular, **AXF** introduces a system of *microcitations* that apply to portions of manuscripts *as well as* data sets. In the publishing context, a microcitation is defined as a reference to a partially isolated fragment of a larger document, such as a table or figure illustration, or a sentence or paragraph defining a technical term, or (in mathematics) the statement/proof of a definition, axiom, or theorem. In data publishing, "data citations" are unique references to data sets in their entirety or to their smaller parts. A data microcitation is then a fine-grained reference into a data set. For example, a data microcitation can consist of one column in a spreadsheet, one statistical parameter in a quantitative analysis, or "the precise data records actually used in a study" (in the words adopted by the Federation of Earth Science Information Partners to define microcitations; see [?]). As a concrete example, a concept such as "expiratory flow" appears in **CORD-19** both as a table column in research data and as a medical concept discussed in research papers; a unified microcitation framework should therefore map *expiratory flow* as a keyphrase to both textual locations and data set parameters. Similarly, a concept such as *2'-C-ethynyl* (mentioned earlier, in the context of transcription errors) should be identified both as a phrase in article texts and as a molecular component present within compounds whose scientific properties are investigated through **CORD-19** research data. In so doing, a search for this concept would then trigger both publication and data-set matches at the same time.

Further discussion on data microcitations depends on how data sets are structured, which is addressed in the next section.

6.3 Research Objects and Data Microcitations

The design of **AXF** assumes that many Research Archives, comprising of multiple publications sharing an academic focus, will also include open-access research data. The primary motivation for publishing research data is to ensure transparency and reusability — open-access data allows readers to verify that scientific/technical claims are warranted, and/or to reuse or incorporate existing data into new research. Open-access data has other purposes as well: data sets, for example, can serve as pedagogic tools helping readers understand publications' concepts experimentally and interactively (a good example is [?], which pairs data sets with its individual chapters to illustrate principles in data visualization). Moreover, the theory informing how data sets are organized can serve as a technical exposition of research principles and methodology. For all of these reasons, open-access data is an increasingly important part of the publishing ecosystem. This means that well-curated archives will often need to prepare data sets for data mining, alongside the preparation of text materials for text mining.

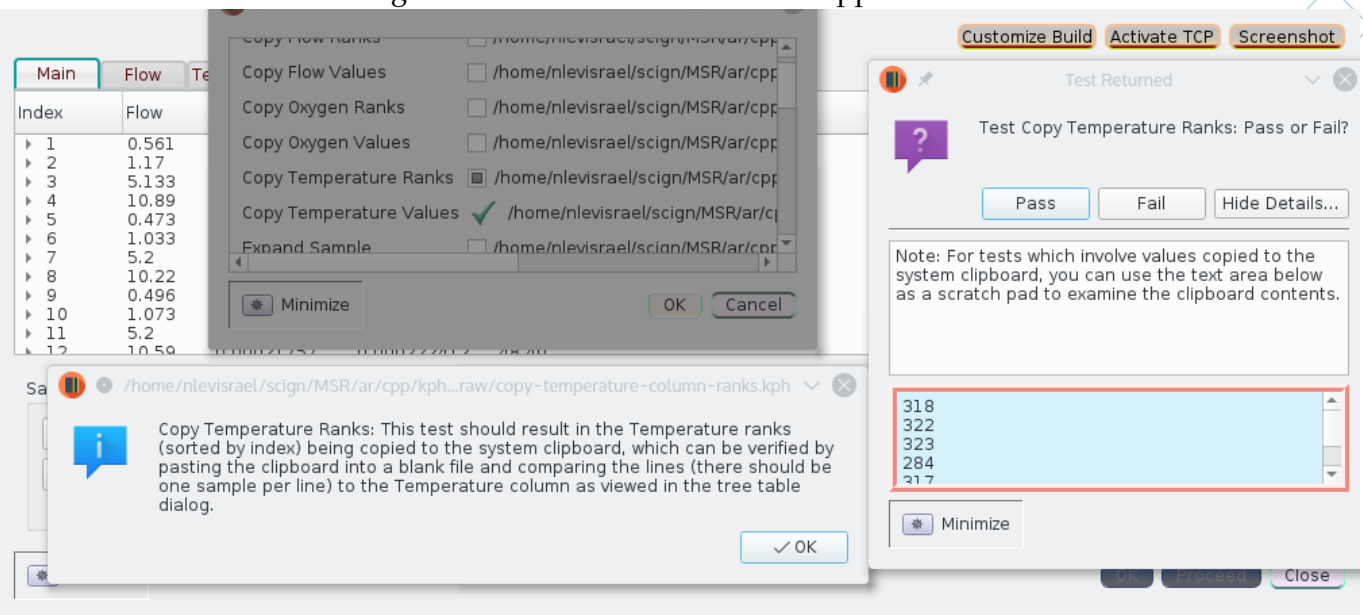
In contrast to text mining, however, it is not feasible, in the general case, to assign one single format (like AXFD or JATS) for all data sets published within an archive. Precisely how data sets can be annotated depends on the data models, programming languages, and analytic methodologies which they utilize. Because this variability prohibits a single data-annotation protocol from being required, AXF adopts a strategy of defining a rigorous protocol for C++ code bases, which can then be emulated by other languages.

As outlined above, data citations refer to parts within a data set — such as individual data records, but also larger-scale aggregates such as table columns or statistical parameters. The complication when defining data citations is that a concept such as a table column, although it may have an obvious technical status as a discrete conceptual unit from the point of view of scientists curating, studying or reusing a data set, does not necessarily correspond to a single coding entity that could be isolated as an annotation target. It is therefore the responsibility of *code base annotations* to provide annotations for computational units — such as data types, procedures, and GUI components — that have an annotatable *conceptual* status relative to the data set on which the code operates. Often this will involve mapping one concept to several computational units (for instance, several procedure implementations).

For a concrete example of these points concerning data citations, consider the data set pictured in Figure 4, representing cyber-physical measurements used to calculate oxygenated airflow. The data-set application (interactive-visualization code deployed within the Research Object) displays tabular data via a tree widget (which functions as a generalized, multi-scale spreadsheet table), with tabular columns expressing quantities — such as air flow and oxygen levels — in several formats (raw measures as well as sample rankings and min-max percentages). Conceptually, these columns have distinct methodological roles and therefore can be microcited; indeed, the application links the columns to article text where the corresponding concepts are presented (see Figure 5). However, the implementation does not introduce a distinct C++ object uniquely designating individual columns. Instead, the individual columns can be annotated in terms of C++ methods providing column-specific functionality. In the current example, these methods primarily take the form of features linked to context-menu actions (copying column data to the clipboard, sorting data by one column, etc.). In general, rather than a rigid protocol for data-set annotations, AXF proposes heuristic guidelines for how best to map programming constructs to scientifically salient data-set concepts.

Defining an annotation schema for data sets can potentially be an organic outgrowth of software-development methodology — viz., the engineering steps, such as implementing unit tests, which are essential to deploying a commercial-grade application. This point is illustrated in Figure 6, which shows a GUI-based testing environment for the data set depicted in Figures 4 and 5. For this data set, the context menu actions providing column-specific functionality are also discrete capabilities which can be covered by unit tests, so the set of procedures mapped to the citeable concept correspond with a set of unit-test requirements. In this data set, these procedures are also

Figure 4: Test Suites for Dataset Applications



exposed to scripting engines via the QT meta-object system. In general, there is often a structural correlation between scripting, unit testing, and microcitation, so that an applications' scripting and testing protocol can serve as the basis for annotation schema. For data sets which use in-memory or persistent databases, evaluable queries against these databases provide an additional grounding for annotations. In general, data-annotation should be engineered on the basis of a dataset applications' scripting, testing, and/or query-evaluation code. However, this is only a heuristic guideline, and AXF does not presuppose any data-annotation scheme *a priori*.