WHITE PAPER

New Database Engineering and Archive Construction Technology to Accelerate Covid-19 Research

LTS (Linguistic Technology Systems) is founded by Amy Neustein, Ph.D., Series Editor of **Speech Technology and Text Mining in Medicine and Health Care** (de Gruyter); Editor of **Advances in Ubiquitous Computing: Cyber-Physical Systems, Smart Cities, and Ecological Monitoring** (Elsevier, 2020); and co-author (with Nathaniel Christen) of **Cross-Disciplinary Data Integration and Conceptual Space Models for Covid-19** (Elsevier, forthcoming).

Team

Principal Investigator Dr. James A. Rodger, Professor of Management Information Systems and Decision Sciences at Indiana University of Pennsylvania.

Administrative Officer Dr. Amy Neustein, founder of Linguistic Technology Systems (LTS).

Contributors

- Nathaniel Christen, software engineer, LTS.
- Professor Amita Nandal, Department of Department of Computer and Communication Engineering at Manipal University, Jaipur
- Professor Arvind Dhaka, Department of Department of Computer and Communication Engineering at Manipal University, Jaipur; recently visiting scholar at University of Varna, Bulgaria
- Professor Todor Ganchev, Vice Rector of Research at University of Varna, Bulgaria

Executive Summary

LTS is building a pair of SARS-CoV-2/Covid-19 repositories to advance both Covid-19 research and new methodologies for data-integration and **AI** research in general. The first of these repositories, the "Cross-Disciplinary Repository for Covid-19 Research" (**CR2**), is *empirically-focused* — aggregating published data sets into a common research platform so as to promote Covid-19-related data mining. The second of these repositories, "**AI** Methodology and Conceptual Space Theory" (**AIM-Concepts**), is *methodology-focused* — providing a collection of code libraries implementing techniques applicable both to data integration and to Artificial Intelligence research. **AIM-Concepts** prioritizes analytic methods and data representations — such as hypergraphs, fuzzy sets, and conceptual spaces — which have broad applications in software engineering as well as **AI**. Both **CR2** and **AIM-Concepts** are companion resources to the forthcoming Elsevier volume (authored by the LTS team) titled *Cross-Disciplinary Data Integration and Conceptual Space Models for Covid-19*. In this volume, we will provide concrete examples of Covid-19 data integration/analytics by examining data sets included in the **CR2** repository; we will likewise demonstrate analytic techniques by examining computer code included in the **AIM-Concepts** repository.

AIM-Concepts, with respect to methodology, will focus on Conceptual Space Theory, which is a valuable unifying framework connecting to both fuzzy sets as an **AI** analytic strategy and to hypergraph models as a data representation strategy. In addition to **AIM-Concepts** providing **AI** code libraries based on Conceptual Spaces, **CR2** will likewise feature conceptual space models,

applying this paradigm as a framework for describing research data. To this end, we will be introducing an updated version of Conceptual Space Markup Language (CSML) — expanded in order to serve as a general-purpose dataset-description language. CR2 and AIM-Concepts will be connected by a new hypergraph database protocol, which we are calling "Transparent Hypergraph Query Language" (THQL). The CR2 archive will include an implementation of this protocol used to aggregate CR2 data into a common format. Likewise, the methodology presented in the AIM-Concepts archive will be examined via demonstrations of how techniques and algorithms of the AIM-Concepts libraries can be applied to data hosted in a THQL database. The THQL technology has many concrete applications outside the context of Covid-19, penetrating vertical markets such as pharmaceuticals, manufacturing, fintech, healthcare, educational software, and bioinformatics.

The principle goal shared by both CR2 and AIM-Concepts is to provide a *common* programming infrastructure which facilitates the implementation of algorithms and GUI components that synthesize data across different scientific fields and methodologies. This common infrastructure is based primarily on Qt — a C++ application-development framework — and secondarily on established scientific/medical code projects such as CaPTk (the Cancer Imaging Phenomics Toolkit, from the Center for Biomedical Image Computing and Analytics) and ReproZip (a tool for packaging scientific software dependencies). In addition to utilities and build tools, the programming core for CR2 and AIM-Concepts will include C++ libraries to read and manipulate data in formats commonly used for clinical, diagnostic, imaging, sociodemographic/sociogeographic, and medical-outcome reporting. Moreover, LTS will provide a new library, called "MOSAIC," to help build applications for visualizing and reusing data sets.

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 to a repository including multiple data sets. Data-set applications can, if desired, be developed as *notebooks*, with features inspired by "computational notebook" technologies, such as Jupyter and Kaggle. The MOSAIC libraries include code for a Qt Creator plugin which allows the Qt Creator IDE (Integrated Development Environment) to be used as a computational notebook. In addition to serving as standalone applications in themselves, MdsX notebooks can be embedded in other applications; for instance, in scientific software.

MdsX is paired with MOSAIC *portal*, a code library for hosting data sets and publications, and MOSAIC *plugins*, which allow MOSAIC to be used in pre-existing applications. 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 structured 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 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. When embedded in host applications, the "MOSAIC plugin framework" (MPF) allows MOSAIC plugins to send data back and forth to one another, thereby allowing their host applications to interoperate.

In addition to user-interface code, MOSAIC notebooks will generally include code for reading data from a file (or potentially from a database or web resource). Each notebook may therefore depend

on a code library managing specific file types and data formats. Ideally, these libraries should be distributed in source-code fashion with the notebook code itself. To facilitate the construction of notebooks, **CR2** will provide parsers for several file formats commonly used in biomedicine, such as **OMOP** (from the Observational Medical Outcomes Partnership), **PCORNET** (defined by the Patient-Centered Clinical Research Network), **FHIR** (Fast Healthcare Interoperability Resources), **RADLEX** (Radiology Lexicon), **LOINC** (Logical Observation Identifiers Names and Codes), **FCS** (Flow Cytometry Standard), **PMML** (Predictive Model Markup Language), **ARFF** (Attribute-Relation File Format), and **HDF5** (Hierarchical Data Format version 5).

This paper is intended as an introduction both to the the **CR2** and **AIM-Concepts** repositories and to new technologies, such as **MOSAIC**, which we are developing alongside them. Part I of this paper will outline the repositories in greater detail; Parts II and III will focus on the new technologies; and Part IV will examine specific biomedical areas (such as diagnostic imaging and vaccine/immunology research) from the perspective of these technologies.

Part I: The Covid-19 Repository

Introduction

In an effort to accelerate scientific discovery and therapeutic intervention for Covid-19, LTS is engaged in the curation of two new repositories: the "Cross-Disciplinary Repository for Covid-19 Research" (**CR2**); and the "**AI** Methodology and Conceptual Space Theory" (**AIM-Concepts**). The purpose of these repositories is to centralize *disparate* Covid-19-related data and code into a common research platform. This code and data, as much as possible, will be marshaled into a common, hypergraph-based representation format, and a common **C++**-based programming environment. In building the **CR2** repository, LTS will develop and demonstrate new database engineering and dataset/repository construction technologies. These technologies have deep market penetration-potential in many areas, including, but not limited to, pharmaceuticals, manufacturing, fintech, healthcare, educational software, and bioinformatics.

Some of the **CR2** code/data will be hosted on GitHub at Mosaic-DigammaDB/CRCR (for data aggregation) and Mosaic-DigammaDB/LingTechSys (for code previews). The latter repository includes preview code sampling database engine features, such as the logic for constructing a database in shared memory, encoding data types for persistence, and so forth. The data management tools developed for the **CR2** repository have a broad range of use-cases, and can be customized for different projects. Companies or research groups interested in a more substantial code preview are invited to contact LTS to discuss their projects and requirements in greater detail.

The main challenge when curating a data repository such as **CR2** is reconciling heterogeneous data formats. In response to this challenge, LTS has focused on hypergraph-based data models which can unify many different information structures into one common structure. In particular, **CR2** will introduce a special "Hypergraph Exchange Format" (**HGXF**) which can take the place of disparate tabular or graph file formats (comma-separated values, numeric python, spreadsheets, graph-network serializations, etc.), so as to merge data sets into a common *machine-readable* archive. In addition, **CR2** will introduce a new protocol for engineering hypergraph databases, called "Transparent Hypergraph Query Language" (**THQL**). Databases conformant to the **THQL** model will be able to export data in hypergraph-based formats, thereby generating data sets which can be used as published, citable Research Objects. **CR2** will demonstrate **THQL** via a new database engine called "DigammaDB" (or **ƓDB**) which serves as a "Reference Implementation" for **THQL**. In **CR2**, **ƓDB** functions as a prototype and reference example for **THQL**, used to curate data sets before their final form is exported into the main data repository. LTS can also customize commercial versions of a **THQL** engine tailored to the requirements of individual projects.

THQL is designed with a priority on application development. In particular, any instantiation of **THQL** should provide data persistence capabilities through (as much as possible) self-contained code libraries that can be included in source-code form within an overall application. **THQL** is designed to integrate seamlessly with native, desktop-style standalone applications. In short, **THQL** represents an unprecedented combination of native desktop-style software development and hypergraph database engineering.

Complementing **THQL**'s application-development focus, **CR2** will also introduce "Dataset Creator," (dsC), a new tool for curating research data sets. The main feature of Dataset Creator is its use of native software components (called "Dataset Applications"), allowing researchers to view, manipulate, and reuse research data. In sum, the typical Research Object built with dsC will include self-contained source code implementing a customized desktop application providing access to the accompanying data set. In addition to **GUI** code, each Dataset Application will supply "data-access" code libraries for parsing the raw data-set files, so as to obtain the information visualized within the **GUI** classes of the Dataset Application. These data-access libraries offer machine-readable access to the raw data, permitting subsequent researchers to reuse the data-access software libraries so as to transform, filter, or analyze the published data in the context of replication studies and/or novel research projects.

Development of **THQL** and dsC is concomitant with **CR2**; the **CR2** repository will provide a practical test-bed for validating this new technology. Accordingly, the following sections will describe **CR2** in greater detail, followed afterward by sections offering more information about **THQL** and dsC.

The Cross-Disciplinary Repository for Covid-19 Research

The sudden emergence of Covid-19 as a global crisis has cast a spotlight on computational and technological challenges which, in the absence of a catastrophic pandemic, would rarely rise to public attention. In particular, an effective response to the dangers of SARS-CoV-2 requires coordinated policy making integrating diverse modes of scientific inquiry. Genomic, biomolecular, epidemiological, socio-demographic, clinical, and radiological information are all pertinent to Covid-19. In this environment, it is important that the empirical foundations for expert recommendations — which in turn drive public policies of enormous social and economic consequence — be transparently documented and critically examined. The proper synergy between government and science depends on data centralization: given the gaps in our current Covid-19 knowledge, it is understandable that different jurisdictions will craft responses to the pandemic in different ways. There is no central authority with sufficient epistemic force to legitimize homogeneous mandates across the entire country. However, such policy differences should be a consequence of alternative interpretations of scientific knowledge or the diverse needs of local communities — rather than being a haphazard consequence of governments working with divergent, competing, and poorly integrated data.

The current administration, along with numerous corporate and academic entities, has clearly recognized the need for a more centralized paradigm for sharing Covid-19 data. For example, the White House spearheaded a scientific initiative to develop **CORD-19**, an open-access corpus of over 46,000 peer-reviewed publications related to Covid-19, which were transformed into a common machine-readable representation so as to promote text and data mining. Similarly, large institutions such as Google, Johns Hopkins, and Springer Nature have all implemented some form of coronavirus data-sharing platform targeted to both scientists and policy makers. However, these two aspects of the corporate/academic contributions to Covid-19 data sharing (exemplified by the **CORD-19** White House initiative and by institution-generated portals, respectively) have been incomplete, for opposite but complementary reasons. Specifically, **CORD-19** is highly structured and tightly integrated, but it focuses primarily on text mining and scientific documents, not *research*

data. While it is possible to find data sets about Covid-19 through **CORD-19**, the techniques to do so are both cumbersome and non-scalable. On the other hand, projects such as the Johns Hopkins coronavirus "dashboard" provide accessible data sets, yet these projects are isolated and do not offer the level of structure and integration evinced by **CORD-19**. In short, an optimal Covid-19 research platform would merge the structural text-mining rigor of **CORD-19** with the data-centric focus of isolated projects that share Covid-19 data with the scientific community, policy makers, and the general public.

The benefit of **CR2** is that it can accelerate Covid-19 research by (1) pooling a diverse collection of data sets into a single resource which scientists can utilize; (2) serving as the prototype for larger research portals that can aggregate new Covid-19 data that will emerge from hospitals, labs, and academic institutions in the future; (3) formalizing a framework for aggregating patient narratives to accurately capture first-hand subjective symptomatology of the patient suffering from Covid-19; and (4) accelerating the implementation of novel data-integration and software-development technologies which can contribute to scientific progress vis-à-vis Covid-19 in particular, and biomedical/scientific computing methodology in general. These principles, in turn, will shape the design of **CR2**. An ideal data-sharing ecosystem should merge data from multiple sources, but should do so in a fashion which yields a machine-readable totality, analogous to **CORD-19**'s structuration with respect to text mining. The merit of **CR2** therefore lies not only in the data which it will encompass but also in novel technology that it will concretize for constructing data repositories adhering to these goals — aggregating data, but also instantiating novel data-integration and database engineering strategies.

Given these varying goals, **CR2** can provide value at different scales of realization. Relatively small data sets serve several scientific and computational purposes: (1) they can provide researchers with a mental picture of how data in different disciplines, projects, and experiments is structured; (2) they can serve as a prototype and testing kernel for technologies implemented to manipulate data in relevant formats and encodings; and (3) they can lay the foundation for data-integration strategies. For example, when designing a representation format and/or implementing code to merge different data formats into a single structure (or meta-structure), it is useful to work with small, representative examples of the data structures involved, so as not to complicate the integration logic with computational details solely oriented to scaling up the data-management logistics. As a result, **CR2** can provide a testbed for implementing data-integration technologies which can scale up as needed. To fulfill this mission, **CR2** can aggregate relatively small data sets which have previously been published on academic and research portals, such as Springer Nature, Dryad, and DataVerse. At the same time, a more substantial (and not necessarily fully open-access) Covid-19 data-set collection would also be beneficial to the scientific and policy-making community. Ideally, then, **CR2** will be paired with a larger technology which shares a similar implementational strategy but with different accession paradigms, allowing for an open-ended collection of Covid-19 data which users may selectively access (instead of a single package that users may acquire as an integrated resource). The common denominator in both cases (whether the focus is on relatively smaller or larger data sets) is the importance of deploying novel and contemporary data-integration techniques to centralize Covid-19 research as much as possible. Accordingly, this summary will briefly explain how **CR2** can accelerate Covid-19 data integration on both a practical and technological level.

Methodology for Covid-19 Data Integration

As indicated above, pertinent Covid-19 data is drawn from multiple scientific disciplines. On a technological level, Covid-19 data is documented via a wide array of file types and data formats. This diversity presents technological challenges: if a Covid-19 information space encompasses files representing 25 different incompatible formats, users would need 25 different technologies to fully benefit from this data. In many cases, however, data incompatibilities are merely superficial

— an important subset of Covid-19 data, for example, has a common tabular meta-model, even if the data is realized in discordant technologies (spreadsheets, relational databases, comma-separated-value or Numeric Python files, and so forth). Applying **CR2**'s technology, one level of data integration can thus be achieved simply by encoding tabular structure into a common representation: any field in a table can be accessed via a record number and a column name and/or index. In some cases, more rigorous integration is also possible — for example, by identifying situations where columns in one table correspond semantically or conceptually to those in another table. In either case, it is reasonable to assume that a single abstract data format lies behind surface data-expression in patterns such as spreadsheets and comma-separated values (**CSV**), so that all files in an archive encoding spreadsheet-like data can be migrated to a common model.

Other forms of clinical and epidemiological inputs are often more amenable to graph-like representations. For instance, trajectories of viral transmission through person-to-person contact is obviously an instance of social network analysis. Similarly, models of clinical treatments and outcomes can take graph-like form insofar as there are causal or institutional relations between discrete medical events: a certain clinical observation *causes* a care team to request a laboratory analysis, which *yields* results that *factor* into the team's decision to *administer* some treatment (e.g., a drug *from* a particular provider *with* a specific chemical structure), which observationally *results* in the patient improving and eventually *being* discharged. In short, patient-care information often takes the form — at least conceptually — of a network comprised of different "events," each event involving some observation, action, intervention, or decision made by care providers, and where the important data lies in how the events are interconnected: both their logical relationships (e.g., cause/effect) and their temporal dynamics (how long before a drug leads to a patient's improvement; how much time elapses between admission to a hospital and discharge). These graph-like representations are a natural formalization of "patient-centered" data models.

Using **CR2**'s associated software (for example, importing Covid-19 data sets into a **THQL** database), a higher level of data integration can then be achieved by merging tabular and graph-like models into a single *hypergraph* format. A significant subset of Covid-19 data (or, more generally, any clinical/biomedical information) conforms to either tabular or graph structures; thus it is feasible to unify all of this information into a common framework. A graph-plus-table architecture is generally considered some form of Hypergraph model, and indeed **CR2** adopts a hypergraph paradigm to merge many different sorts of information into a common structure. In particular, **CR2** introduces a new "Hypergraph Exchange Format" (**HGXF**) which can provide a text encoding of many files that, when originally published, embodied a diverse array of file-types requiring a corresponding array of different technologies. **CR2** will include specialized computer code that would enable machine-readability of the **HGXF** files, and use them to create hypergraph-database instances. In short, **CR2** will promote Covid-19 data integration by translating a wide range of files into a common **HGXF** format.[1]

Hypergraph Data Models and Multi-Application Networks

As has been outlined thus far, via the **CR2** technology most Covid-19 data can be wholly or partially integrated into a single hypergraph framework, which accordingly simplifies the process of designing software applications and algorithms to analyze and manipulate this data. Specifically, software components can employ a single code library to obtain, read, consume, and store data, rather than needing to re-implement this logic for a large number of different file formats and/or

[1] **CR2** data sets are not required to compile all files to a hypergraph format; in particular, sciences requiring substantial quantitative analysis — e.g., biomechanics or genomics — express data via encodings optimized for relevant mathematical operations, and have parser libraries optimized for these specific formats. For these files **CR2** will generally provide an **HGXF** encoding supplying data *about* the original file, with information concerning the file type, preferred software components for viewing/manipulating its data, etc., so that the contents of non-**HGXF** files can be indirectly included into the **CR2** hypergraph-based ecosystem.

database models.

Quality software (especially in the clinical and biomedical context) demands a balance between applications which are either too broad or too narrow in scope. On the one hand, doctors often complain that homogeneous Electronic Health Record systems (where every digital record or observation is managed by a single all-encompassing application) are unwieldy and hard to work with. This is understandable, because the clinical tasks of health care workers with different specializations can be very different. On the other hand, doctors also complain about software and information systems which are so balkanized that they must repeatedly switch between different, non-interoperable applications. In short, clinical, diagnostic, and research software should be neither too homogeneous nor too isolated; finding the proper balance between these extremes is, no doubt, a major challenge to the usability of electronic health systems going forward.

Against this background **CR2** demonstrates novel solutions to this problem: it focuses on the dimensions of data acquisition and management that are specific to individual scientific or medical specializations, while also identifying requirements that are consistent across domains. Scientific software generally needs to hone in on the data visualization and analytic requirements of particular disciplines; for example, biochemists use different programs than astrophysicists. However, much of the code underlying scientific applications has nothing to do with these high-level models or theories, but is simply a fulfillment of basic data-management functionality — data storage, accession, provenance, searching, user validation, and so forth. In effect, the computational requirements of scientific and biomedical software can be partitioned into two classes: (1) domain-specific logic which reflects the quantitative or theoretical models of narrow scientific fields; and (2) data-management logistics which can be realized within a central access hub, rather than being re-implemented by each application in isolation.

In short, **CR2** architecture conceives of a central hub responsible for storing data and serving as a common access point — providing the "gateway" where authorized users can gain access to heterogeneous information spaces utilized by an array of domain-specific software applications. Since peer applications would not be directly responsible for data persistence or user identity management, they can focus on their specific data analysis and visualization capabilities. The central hub, serving multiple peer applications, is then a heterogeneous data space managing information from multiple applications while also tracking information about the applications themselves: helping users to identify and launch the software which is most directly relevant to their clinical or research needs at the moment. Meanwhile, because peer applications are jointly connected to a central hub, it is possible to implement scientific workflows where one application may send and receive data from its peers, allowing applications to complement each others' capabilities.

This multi-application networking architecture has precedents in some of the current database and engineering technologies. For example, many hospitals and medical institutions employ some version of a "Data Lake," pooling disparate data sources into a heterogeneous aggregate which is then accessed by multiple client applications. Similarly, Machine Learning and Artificial Intelligence often adopts "software agents" or analytic modules in contexts such as Online Analytic Processing, which again represent semi-autonomous software components sharing an originary data hub. Web applications, too, often act as domain-specific subsidiaries deferring operational requirements, such as user authentication or transaction processing, to a central web service. The limitation of multi-application networks in these existing contexts are that the software agents involved are generally "lightweight," with relatively primitive user-interface design. By contrast, the hypergraph technology introduced with **CR2** will support multi-application networking in the context of more substantial desktop-style scientific applications. In sum, the novel hypergraph technology developed by LTS offers a hybrid of the development methodologies employed for desktop scientific software and those applicable to multi-agent heterogeneous data stores, like a

Semantic Data Lake. To accomplish these goals, CR2 will utilize a new hypergraph database engine, coded in the C++ programming language, which has a unique focus on supporting native GUI applications from the ground up, including persisting application state and storing application documentation within the database itself.

A New Paradigm for Data Sharing and Data Transparency

One exceptional feature of Covid-19 research is the extent of public attention focused on scientific discoveries about the disease. Academic and commercial research teams find themselves in an unprecedented situation where there is unusual pressure to accelerate the Research and Development process, and a concomitant demand for a novel level of transparency and openness. For example, vaccine development protocols are being fast-forwarded to take months instead of years, and information about the development process (such as trial results and scheduling) will likely be shared with the public much more than is standard practice. This new reality, in turn, calls for a commensurate evolution in the technology for public data-sharing.

In conventional biomedical R&D, much of the research data is proprietary, and revealed only in restricted contexts to select parties (such as the Food and Drug Administration). Data which *is* then publicly shared tends to be tied to published research papers in peer-reviewed literature, primarily read by a relatively small, specialist audience. All of this is changing with SARS-CoV-2: companies pursuing Covid-19 R&D (in the context of vaccine trials, for example) are facing pressure to publicly share their results as soon, and as transparently, as possible; and policy makers, scientists, and journalists are no less looking for quick access to research data directly, rather than circuitously through academic publications.

CR2 will introduce the new Dataset Creator technology targeted toward this new environment of direct, transparent data-access (dsC will be discussed in greater detail below). Data sets created via this technology therefore implement the "Research Object Protocol," which mandates that research data be bundled with code allowing scientists to analyze and manipulate the information in the corresponding data set. The Research Object framework was designed by a consortium of academic and governmental entities, such as the National Institutes of Health, to promote a paradigm for data publishing which prioritizes multi-faceted research tools over "raw" data that can be difficult to reuse in the absence of supporting code. In particular, Research Objects should be (as much as possible) *self-contained*, which means that scientists do not need external software dependencies to access and study the data — any special code which is a prerequisite to using this data should be included, alongside the raw data, as part of the Research Object itself.

Dataset Creator enables standalone, self-contained, and full-featured native/desktop applications to be uniquely implemented for each data set, distributed in source-code fashion along with raw research data (dsC Dataset Applications use QT by default to provide native GUI classes, tailored to the relevant Research Object). Adopting such a data-curation method makes data sets easier to use across a wide range of scientific disciplines, because the data sets are freed from having to rely on domain-specific software (software which may be commonly used in one scientific field but is unfamiliar outside that field). In addition, Research Objects composed with dsC can be integrated into Multi-Application Networks because the dataset applications are autonomous native GUI applications that can easily interoperate via QT messaging protocols.

Of course, most of the CR2 data sets are previously-published work composed via older technology. Many of these resources, created with a wide range of software products, predate (or fail to apply) contemporary specifications such as the Research Object Protocol; not every CR2 data set will have the full set of features described in this section. However, CR2 will try to maximize the value of each data set by translating them into a QT-based format — in particular, CR2 will provide QT code for reading HGXF files, as well as a QT-based hypergraph representation library.

Following the data integration methods outlined earlier, much of the CR2 data can be merged into a QT-based framework, which can facilitate the implementation of new, more sophisticated Dataset Applications as the information in CR2 gets reused for subsequent research. CR2 will also include QT-based software, such as a customized PDF viewer, which will help researchers utilize the corpus in its entirety. For example, CR2's PDF viewer will include special code to connect PDF files with data sets via "micro-citations," as discussed in the next section.

Supporting Data Micro-Citations to Improve Machine Readability

The CR2 database engine supports annotating individual components of a database — a technology sometimes referred to as "micro-citation." Data micro-citations are references to integral parts of a data set, such as an individual table, or a single row/record or column in a table. Micro-citations allow these integral parts within the data set to be cited by and linked to publications, for purposes of machine readability and attribution. As an example, preliminary vaccine trials often target a patient cohort selected for demographic or medical criteria matching the population who would most benefit from the vaccine. These criteria for selecting the cohort for the vaccine study are usually described in the texts of the articles. However, these criteria are also identified within the data set by socio-demographic data which is part of the information generated by the trial. By making these connections between criteria discussed in the article and those represented in the corresponding data set explicit, text and data mining can be *merged* as analytic tools targeting a data repository, so that machine reading is able to mine not just article text but the corresponding data.

One reason why micro-citations are important is that they clarify the scientific meaning attributed to data set elements by connecting these elements to scientific concepts and "controlled vocabularies" (such as a list of drug names, diseases, proteins, etc.). For instance, micro-citations allow table columns to be mapped to statistical parameters, enabling their empirical properties (such as min/max values and distribution) to be queried by text and data mining software. Likewise, CR2 enables dimensional and measurement annotations to describe the empirical and experimental significance of the measured or calculated quantities which are stored in a database. Such quantity dimensions model the conceptual roles which particular parameters perform: e.g., the axiation "mJ/cm^2" (millijoule per square centimeter) indicates the intensity of ultraviolet light — any table (or other data aggregate) having a column or field with this dimension is intrinsically associated with observations or experiments pertaining to UV light. Consequently, to locate data sets relevant for research about the clinical uses of antiviral UV radiation, one method is to search for data fields dimensionalized in terms of joule or millijoule per square centimeter. As this example illustrates, data micro-citation — via annotations on data fields, statistical parameters, and table columns — is an important data-mining tool. In short, constructing micro-citations within a database serves two distinct benefits: (1) to aid data mining; and (2) to enable granular links (joining specific parts of articles to corresponding parts of the data set in the data set repository — analogous to hyperlinks between web pages) to be established between publications and data sets, making it *easier* for researchers to find the specific information most relevant to their own research.

Code Libraries and Data Sets for Analytic Methodology

As a companion to CR2, whose essential purpose is to present *empirical* observations concerning Covid-19, we are curating the AIM-Concepts archive, which is focused on code that supports Artificial Intelligence and concrete data-integration methodology. At the core of AIM-Concepts is a collection of code libraries implementing analytic techniques and representations used by AI researchers as well as by software engineers — in particular, different varieties of hypergraphs; fuzzy sets/logic; and conceptual spaces. In order to integrate this code into a common program-

ming framework, the **AIM-Concepts** libraries are ported or modified when necessary, with C or C++ as the primary development language and Qt/qmake as the primary development framework and build system. **CR2** and **AIM-Concepts** will be interconnected by employing **CR2** data sets as concrete examples for demonstrating how **AIM-Concepts** libraries may be used within software applications. Here are some of the methodological approaches (based in part on how Covid-19 research can provide concrete use-cases) which will be important components of **AIM-Concepts**:

Computational Epidemiology Methods in this category generally concern the simulation of disease transmission given a set of initial parameters (such as an infectious agent's reproduction rate and an average degree of person-to-person contact), often concomitant with empirical data, tracking how a disease has spread within some observable subpopulation. The empirical findings, therefore, suggest *a posteriori* which statistical model best fits the actual nature of the disease in question. The accuracy of this analysis depends, in part, on how well the observed subpopulation mirrors the susceptible population as a whole; but it also depends on the accuracy of the mathematical formulae translating initial parameters into projected epidemiological simulations to be compared against *a posteriori* data. As such, concrete expositions of these mathematical frameworks — in particular, computer code implementing the calculations which drive a simulated model — constitute a computational asset in their own right. **AIM-Concepts** will include several code libraries that have been published as tools investigating different quantitative epidemiological models. Although some epidemiological simulations rely primarily on mathematical equations, most epidemiology libraries internally use graph-based models, often employing weighted graphs where edges denote, for example, the probability of viral transmission between people in close contact. As a result, distinct epidemiological models can sometimes be merged into a common analytic framework, using custom quantitative algorithms that are composed within a common graph-traversal framework.

Event Modeling Event modeling, sometimes called "Entity-Event Modeling," is an emerging data-analytic trend which focuses on events, rather than objects, as the most fundamental form of observation within a data space. Conceptually, the rationale for this paradigm is that every property which is attributed to some object can be tied to a specific event wherein the given property was measured, observed, discovered, or inferred. For instance, asserting that a patient is *infected* with SARS-Cov-2 implies that a *test* for SARS-Cov-2 was positive. Focusing attention on the *event* (viz., the test and its result) allows the data model to accrue information in a more detailed manner: in the case of SARS-Cov-2 tests, the relevant testing method/kit used; false positive/negative rates in the population; the time ellapsed between the onset of symptoms (if any) and the test being ordered and the interval until the results are provided; the observations or factors (such as the subject's exposure to an infected family member) which caused health care providers to order the test; and so on. Many of these details might be included in a conventional database as well; however, focusing on events allows the relevant information to be obtained more easily in that the event model supplies a temporal and operational organization for the underlying information. Event models are especially useful when temporal sequencing and duration are important considerations for uncovering scientific facts about the phenomenon being studied. In the context of infectious diseases, intervals such as the length of time between exposure and contagiousness, length of time between exposure and the appearance of symptoms, or the duration of hospital stays, are essential to our understanding of the disease's biology.

Fuzzy Sets and Fuzzy Logic Mathematically, fuzzy logic is often conceived in terms of replacing a simple binary logic ("true"/"false") with a more complex multi-valued logic, wherein properties may be true or false to varying degrees. Thus, on the surface, this introduces quantitative models (such as the calculation of the conjunction or disjunction of fuzzy predicates) as a substitute for purely logical reasoning. In practice, however, fuzzy sets often have the opposite effect: this theory gives rise to methods which can simplify empirical analyses by *eliminating*

quantitative formulae — in particular, parameters of a continuous variable. Fuzzy set models tend to simplify data spaces by collapsing continuous quantities into discrete cases (e.g., *low*, *medium*, *high*), or grouping objects into similarity clusters. Via these operations, data models that depend on computationally intensive mathematical variables can be replaced by qualitative models, often representable in graph form (where graph edges may designate membership in a prototype-class, or the evolution of some observable according to several property classes, each of which collapses a spectrum of granular cases into a single prototype). Fuzzy methods can be shown to be effective simplifications of complex data models by demonstrating that analyses conducted via qualitative reconstructions of a data space, for a given set a observations, are comparable to results obtained from more quantitative methods and/or are a good fit to empirical data. Research data sets which emerge from fuzzy methods can be qualitative models derived from quantitative data spaces, as well as code libraries which perform the relevant transformations.

Conceptual Space Theory Conceptual spaces, which have some similarities to Fuzzy Sets, have likewise been proposed as a paradigm for modeling both scientific knowledge and Artificial Intelligence. Conceptual Space Theory is rooted in cognitive and linguistic investigations of how humans formulate and understand concepts (extending to scientific theories, and in particular how we "process" scientific information to infer facts about the world). Conceptual spaces have, therefore, been proposed as a language for representations or descriptions of scientific knowledge, with an emphasis on how scientific models are built up from individual conceptual parameters (such as points in space/time or notions of length, heat, speed/acceleration, electric charge, etc.). As scientific ideas become formalized, our intuitive conceptualization of spatial or observational quantities gets translated into physical, mathematical, or statistical details: scales and units of measurement, observational value ranges, statistical levels (Nominal, Ordinal, Interval, Ratio), and so forth. Formal implementations of conceptual spaces, therefore, focus attention on the dimensional and measurement properties of data parameters (e.g. scales/units of measurement) — information which is usually not made explicit in the publishing of data sets — and on how these parameters aggregate into conceptual units. Such aggregates include things like how two geographical coordinates that define a geospatial location, in Geographical Information Systems; or how shape and color, in combination, characterize visible objects, such as in image segmentation. One conceptual-space represenation is Conceptual Space Markup Language (**CSML**), which **CR2** will update in order to document scientific parameters within data sets. Other conceptual-space models and analytic libraries have been developed in the context of Artificial Intelligence, where the goal is to simulate the patterns of human conceptualization within **AI** engines; this work can then be incorporated into the **AIM-Concepts** repository.

Cognitive Discourse Analysis and Conceptual Role Semantics Cognitive discourse theory is similar to Conceptual Space Theory in investigating the overlap between conceptualization and scientific knowledge — or, more generally, all of our observed facts or empirical beliefs as they are encoded in language. Although it is obvious that most of our language is based in concrete beliefs about the world around us, this basic linguistic principle is not usually captured with full rigor within formal reconstructions of linguistic expressions. The essential paradigm in cognitive linguistics is the notion of cognitive *grounding* — that is, how every object and event included in a sentence connects to the speaker's fundamental understanding of the situation around them, and its relevant facts and observables. While all linguistic theories acknowledge grounding, cognitive analysis develops a detailed theory of how grounding works in all of its aspects: there are multiple dimensions of grounding, because we connect objects/events to situations in many different ways. For example, objects relevant to a given event play distinct empirical roles (the *agent*, which *causes* something — some change — to happen; the *patient*, which is thereby changed/affected; and potentially secondary participants such as the instrument by which the change is effected; the "benefactor," i.e., the object for whose benefit the change is caused, etc.).

These conceptual differences are rigorously treated within Conceptual Role Semantics, which shares a similar philosophical orientation to Cognitive Discourse Analysis. As a practical tool for analyzing language, these two methodologies provide a matrix of classifications for objects and events (and their corresponding linguistic units) in terms of the situational background where every object and event (in a given discursive context) is perceived. Formally, these classifications then provide a layer of annotation which capture linguistic details at a cognitive level more rigorous than usually found in Natural Language Processing. **AI** methods in this field focus on automatically identifying cognitive-discursive patterns in language, although data sets can also be formed by manually annotating language samples according to cognitive-discourse and conceptual-role vocabularies. In either case (whether via manual or **AI**-driven annotations), Cognitive Discourse Analysis represents one emerging technique for the representation of natural-language assets — such as Patient Narratives — for the purposes of applying advanced text/data mining strategies (in the case of Patient Narratives, to extract critical patient symptomology often buried in circumlocutory discourse).

Adding Patient Narratives to Covid-19 Data

In addition to aggregating published data sets, **CR2** may be used as a repository for collecting new Covid-19 information. With that in mind, we are prioritizing the design of a standard for storing and accessing natural-language text representing patients' subjective symptom descriptions, which is quite useful for diagnostic/prognostic assessments of patients infected by Covid-19.

Just as **CR2** envisions a curation of published data sets for data mining to improve machine-readability of Covid-19 research, LTS also sees the benefit of a repository of patient narratives prepared for text mining, to improve machine readability of the open-ended symptom descriptions offered by patients. While **CR2** does not need to specify how these narratives should be collected, it will implement a common representational format so that patient narratives can be pooled, similar to to how **CORD-19** research texts are merged and encoded with a system that permits annotation.

In modeling patient narratives, this technology will be oriented toward the scientific-computing ecosystem outlined in the previous section. In particular, we assume that **GUI**-based desktop applications will be the primary instruments for data collection and analysis; this means that the encoding of patient narratives may, at times, need to be paired with **GUI** or multi-media content. For example, the software for patients to submit medical history information could also allow them to pair (text-form) narratives with graphics indicating the location of their pain or discomfort. Furthermore, the software could allow narratives to be accompanied by an audio file where patients could cough/speak into a microphone. Given this range of possible inputs, patient-narrative encodings must be flexible enough to include diverse multi-media content.

As described earlier, an information space adapted for multiple peer applications should encompass capabilities for saving application state (the current visual appearance of the program), which includes features for modeling instances of **GUI** classes. This technology provides the necessary infrastructure for managing patient narratives. For example, consider a multi-media intake form where patients may describe symptoms by placing icons (representing pain or discomfort) against anatomic silhouettes (head/body, back/front, extremities, and so forth). As patients use such a multi-media form, **GUI** application state corresponds to the patient's subjective symptomology; in this way the graphics-based representation of symptoms could then be incorporated into the overall patient narrative. This is an example of how application-persistence logic can be marshaled to the related project of curating patient narratives.

Part II: The MOSAIC Data-Set Explorer and Computational Notebooks

Data Sets and Data Publishing: the current picture

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 interoperability 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.

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 **API**s, 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 ecosystem 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 Experience 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 di-

rectly 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.

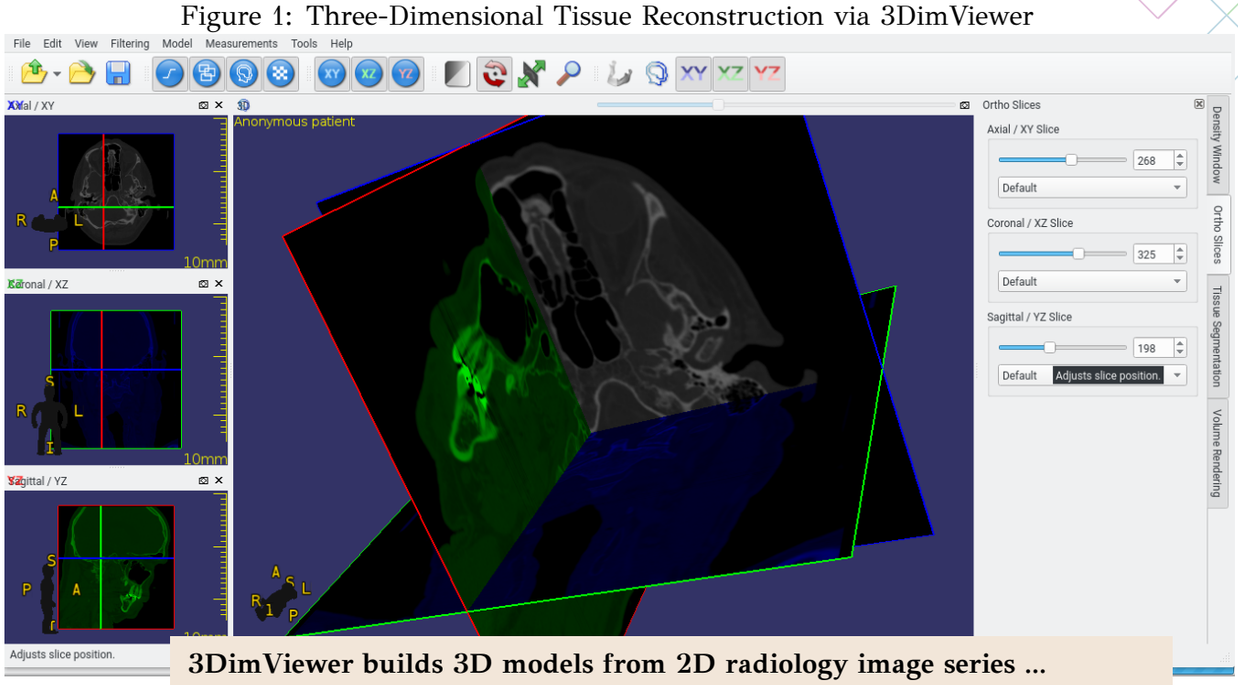
To be sure, scientists can sometimes "package" their experimental and analytic methods into a coherent framework, such as a JUPYTER notebook, which serves as both a demonstration and a precis of their research work. Indeed, tools such as JUPYTER (which packages code, data, and graphics into a self-contained 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 or "notebook"-oriented frameworks like JUPYTER is not truly 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 **MdsX**, which is used to implement *native*, *desktop-style* applications 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 incorporates the important ideas from contemporary data publishing (including ideas originating in the cloud context): workflow models, notebooks, access to publishers' APIs, etc. In short, MOSAIC 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** components are 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.

The overall MOSAIC framework, then, spans three different technical areas. MOSAIC Portal comprises server-side logic for hosting data sets, publications, and cloud services utilized by native MOSAIC applications. MOSAIC Plugins are extensions to larger applications allowing data sharing between plugins and MOSAIC Portal services (and between plugins themselves, embedded in distinct applications). Finally, **MdsX** applications and notebooks can be either standalone or embedded; in the latter case an **MdsX** plugin can also serve as an **MdsX** notebook, if it provides a GUI structured according to the basic **MdsX** design.

Whether or not **MdsX** plugins serve as notebooks, they can be useful enhancements to host applications, allowing for more robust data-sharing and multi-application networking. MOSAIC

Figure 1: Three-Dimensional Tissue Reconstruction via 3DimViewer



3DimViewer builds 3D models from 2D radiology image series ...

plugins, for example, would allow applications to send and receive data using the **HGXF** format; and also to support workflows described via the "Hypergraph Multi-Application Configuration Language" discussed below. These workflows allow different applications' capabilities to complement one another; a simple example is shown in Figure 1, where a **3D** tissue model is built from a **2D** image series, and Figure 2, where the same **3D** model is exported to a **3D** graphics application. Using **MOSAIC** plugins allows the file transfer between the two applications to be performed automatically, rather than requiring users to manually save and then reopen the file.

Although minimal **MOSAIC** plugins provide some useful functionality, the main goal of **MOSAIC** is to support more complex, **GUI** based interactive notebooks which can be plugged in to other applications as well as run standalone. These notebooks will be discussed in the next section.

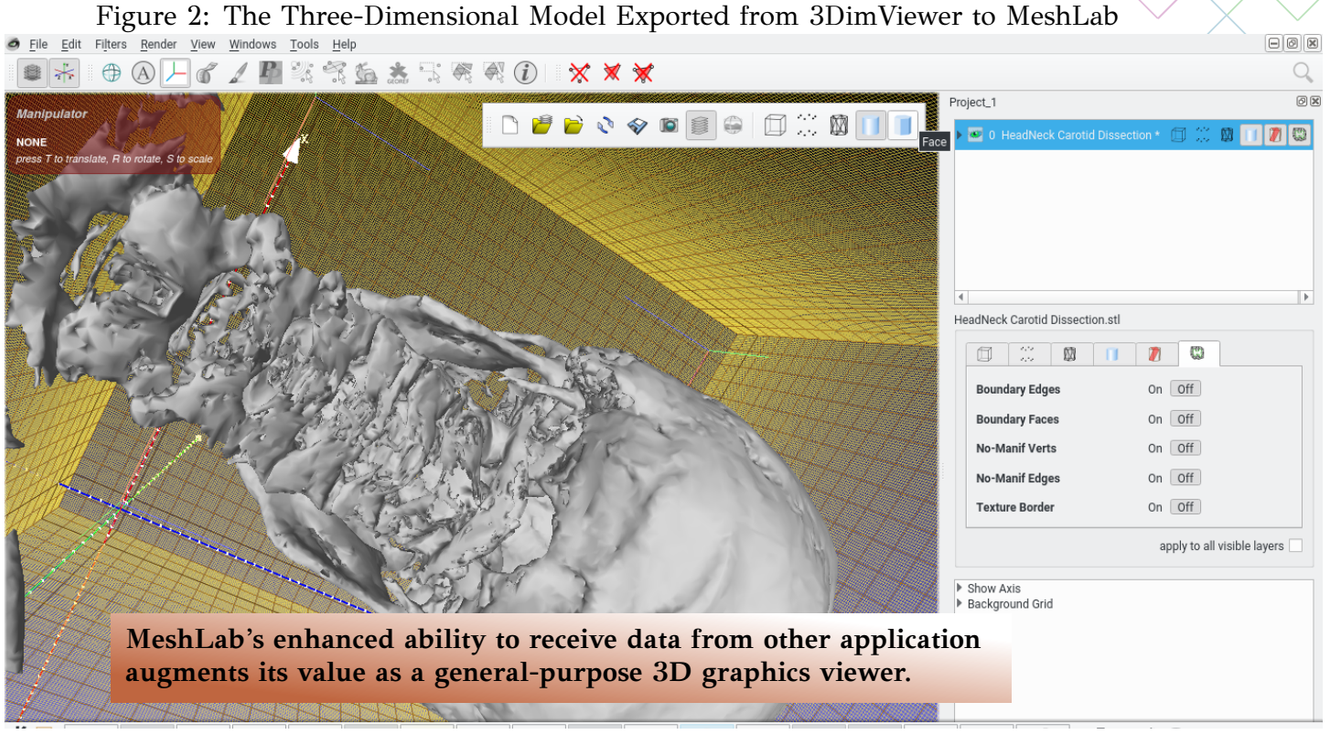
The Structure of MdsX Notebooks

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, computational 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 —

Figure 2: The Three-Dimensional Model Exported from 3DimViewer to MeshLab



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 "**REPL**s 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 degree 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** (and **JUPYTER**), 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

Figure 3: A graphics-based view (foreground) and object-based view (background)

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 **JVM** 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, **CHAISCRIPT**, 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 **ECL** (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. 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 thereby also become part of the notebook content, able to be visualized in their own frame. Notebooks in general, accordingly, can have four varieties of frames (graphics views, navigation 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**, **JPEG**, **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 3 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 and JKQCustomPlotter libraries. 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 would 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**, **JPEG**, **TIFF**, and **DICOM**) applications need to specify details 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 software — 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 a 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-dimensional 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.

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 accordingly be exported or saved, alongside (and as an extension of) notebook state.

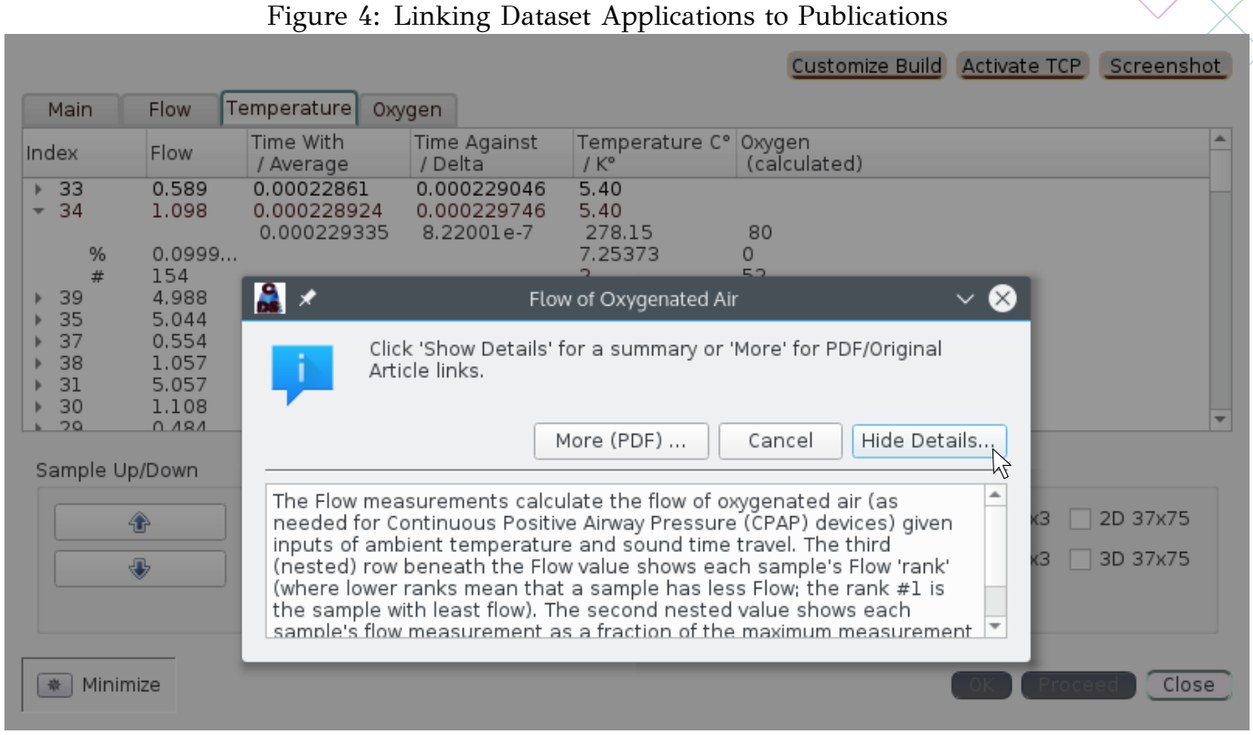
Analytic operations available through an image-based notebook may be provided by the notebook itself, or by a host application where MdsX is embedded. In the latter case, the notebook needs to construct the proper calls to the host application, using the meta-procedural controller as a bridge to ambient 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

Figure 4: Linking Dataset Applications to Publications

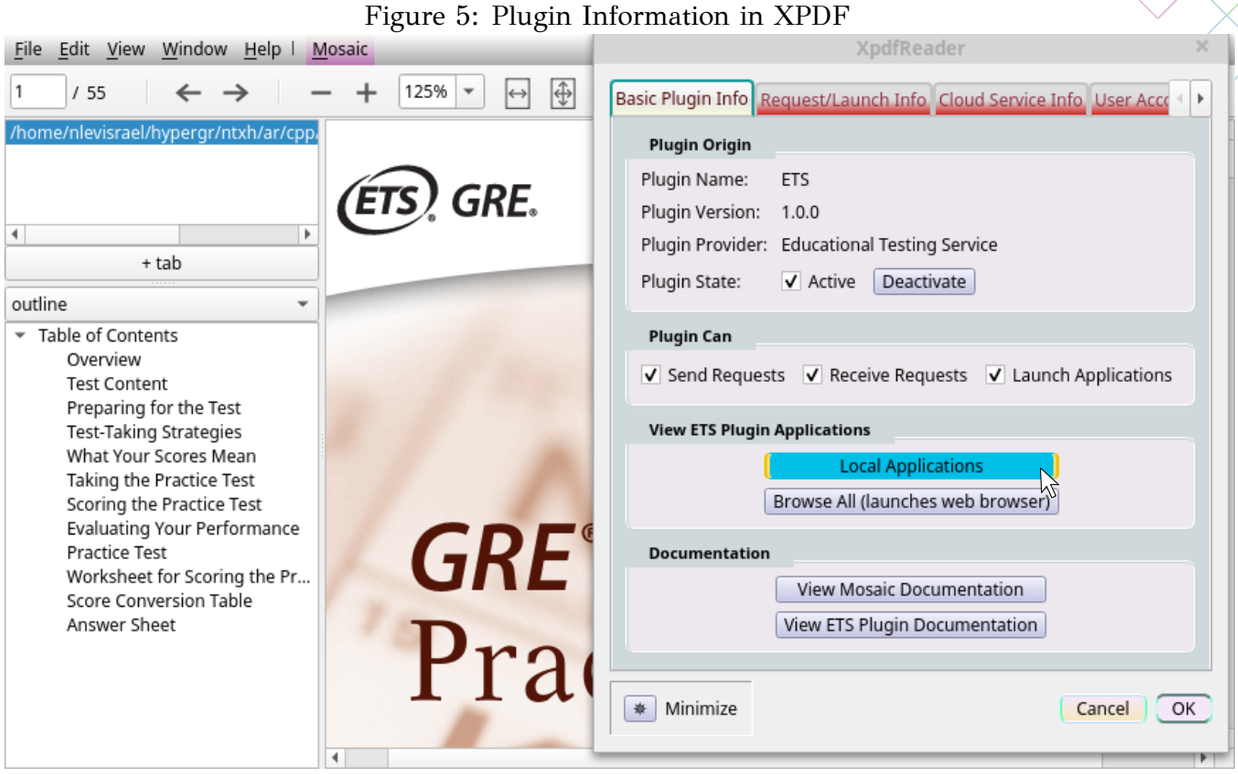


complete "marking," can lead to the meta-procedure being "fired." Each preliminary stage — that is, each pre-firing addition to the execution context — may in turn be generated by events originating elsewhere in the application (canonically, the navigation panel), and the meta-procedural controller should 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 3 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 3. As demonstrated in Figure 4, 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.

This specific genre of annotations — applying to meta-procedure and visual objects which are linked to scientific concepts — is included within a broader **MOSAIC** annotation system, described

Figure 5: Plugin Information in XPDF



below. A rigorous overview of this annotation framework depends on describing the data-structuring protocols recognized by MOSAIC, which are reviewed in the next section.

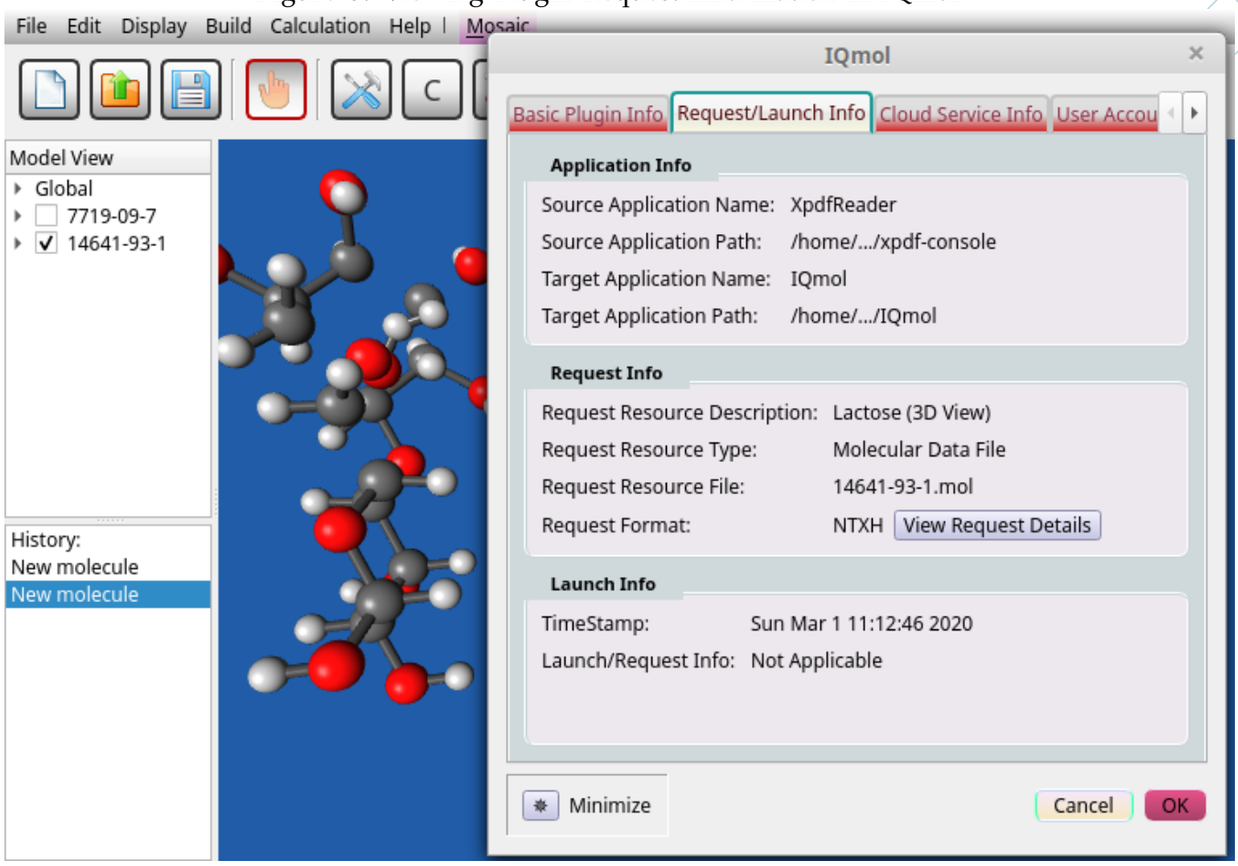
MdsX Data Controllers: Viewing Data Files in MdsX

The data view and controller in MdsX notebooks are responsible for displaying textual serializations of data structures, particularly the specific data sets that form the core of a data-set application. Raw data, of course, may be serialized in many different formats, and spread over multiple files. However, MOSAIC natively recognizes its own data format based on hypergraphs (HGXF) and provides a light-weigth database engine (CDB) for persistent storage of notebook state. Thus, HGXF can be used to serialize raw data viewed in a MdsX notebook as well as overall notebook/application state. Both HGXF and CDB will be discussed in Part III.

Apart from simply showing the text contained in a file, data views should provide information about the file type and format — in particular, should indicate the code libraries which are being used to parse the serialized files. In the ideal case, an MdsX notebook will link against (or include) a C++ library to read files in the associated format, and will also provide C++ classes to hold deserialized data. In simple cases, each file handled by an MdsX data controller corresponds to a list of C++ objects of the same type; in more complex cases the data involves multiple types. In either scenario, the C++ classes defining these de-serialized objects are distinct from those used to parse the files, but an obvious connection exists between these two groups of classes. The data controller is then responsible for documenting these inter-class connections, and also for helping users learn about the relevant classes if they so choose. Assuming an MdsX notebook is embedded in QT Creator, this can include passing a message to the QT Creator application requesting that the relevant C++ files be loaded in to the file-view pane.

In addition to providing information about data files themselves, MdsX notebooks may find it appropriate to provide data provenance summaries. This is illustrated in Figures 5 and 6, showing two applications using a MOSAIC plugin to interoperate. Figure 5 demonstrates how users may view basic plugin info. Figure 6 demonstrates a similar dialog box, but one which is specifically relevant to tracing data provenance; the dialog shows information about the file or files sent between the applications.

Figure 6: Viewing Plugin Request Information in IQmol



When sharing data between two different applications, it is prerequisite that each application (called "end points") have the capability to read the data/file format involved. This can be confirmed by checking that both end points include the same code libraries (for special-purpose data formats) or by translating the data to a generic format such as **XML** or **JSON**. Many data formats can be encoded in multiple ways, some using special-purpose parsers and some based on **XML** or other canonical resource types. A flexible data-sharing protocol should therefore leave open the possibility that several "rounds" of communication may be required to determine how the sending application should encode the information to be transferred, so that the receiving application will be able to decode it.

Given these considerations, applications and plugins which seek to support multi-application networking should, for the most flexibility, support data sharing via both specialized, narrowly-designed data formats and generic, multi-purpose formats. One rationale for using hypergraph-based data serialization is that hypergraphs present a common meta-model able to represent almost all computationally meaningful data structures; as a result, hypergraphs are (theoretically) superior to both tree-form constructions such as **XML**, and tabular conventions such as **CSV** and **SQL**. Accordingly, a maximally flexible data-sharing protocol would include the option of allowing hypergraph-based formats (such as **HGXF**) as a fallback data encoding option, when it cannot be established that both endpoints of an exchange share a more specific data-format library.

Whether the applications are using more generic or more specific serialization formats, data exchange also requires that each application implement the requisite logic to deal with the data once it is deserialized. For each data format, then, one can formally or informally define not only how the associated data structures are serialized/encoded, but also specify what is necessary for a code library which properly handles information from this data once it is deserialized/decoded. This is the kind of information that would be presented to the user via **MdsX** data views, along with the raw data. It is also information that may need to be structurally represented for database or workflow engines. Consider the case of a very heterogeneous data store, such as a Semantic Lake. Such an information space is essentially a decentralized database capable of storing many different kinds of information, used in turn by numerous different applications. In general,

individual applications will only understand and interact with smaller parts of the overall data space: each aggregate unit of data is associated with a set of requirements that applications must satisfy so as to properly manage that particular information. For these reasons, properly modeling such application requirements is an important aspect of heterogeneous database engineering.

A defining feature of *heterogeneous* data stores, as compared to more *homogenous* database engines, is that no single schema or technological infrastructure can directly represent all of the information contained in the heterogeneous space. This basic point is consequential for different aspects of the associated database engineering, such as how to efficiently save, validate, and query all the data in the information space. In conventional database theory, for instance, it is assumed that a single query engine will interact with all data in a database, and the database's logical and storage schema are constructed so as to optimize query evaluation (to get the right results, quickly). When considering *heterogeneous* data spaces, however, we have to anticipate cases where no single query-evaluation strategy (or set of algorithms) will be natively applicable to every aggregate data unit. Instead, query evaluation may need to be performed as a workflow spanning multiple software components, even multiple applications, each performing their own filter, transform, or "reduce" (in the map-reduce sense) operations. There is therefore an engineering overlap between *query evaluation over heterogeneous data space* and *multi-application workflows*. The database engineering technologies discussed in Part III attempt to make this connection explicit by integrating database technology with representations of workflows and multi-application networks, specifically through a newly designed configuration language (HMCL), discussed in the next section.

Part III: Hypergraph Models for Database Engineering and Multi-Application Networking

Hypergraph Multi-Application Configuration Language

The purpose of HMCL is to represent multi-application networking protocols with greater procedural detail and specificity than provided by traditional workflow or Interface Definition languages. Within the scope of multi-application workflows, a good starting point are the protocols adopted by CAPTK, and then by considering analogous protocols connecting semi-autonomous software agents in other contexts: PARAVIEW extensions, QT Creator Plugins, OCTAVE scripts, and ROOT modules, for example (ROOT referring to the physics platform developed at CERN) — restricting attention to the QT/C++ ecosystem. Further afield, a similar sense of semi-autonomy can be found in hybrid GUI and scripting platforms such as JUPYTER (in the Python context), KAGGLE, and SECO. These platforms are familiar to data visualization and machine-learning engineering (where a script may analyze data and a GUI component follow up by presenting a chart or dataplot summarizing the analysis), as well as other quantitative domains outside the natural scientists (in financial services, for example, scripting formats such as MQ4 execute investment strategies while associated GUI components present finance-related visual objects, such as candlestick charts.) Similar workflow models have been developed in theoretically-informed frameworks such as VISSION, which is paired with programming constructs such as "metaclasses" and "dataflow interfaces".[2] The common element in all of these systems is the presence of a central GUI application whose capabilities are enhanced by customizable extensions with their own analytic and/or GUI features: these extensions are neither fully autonomous applications nor simple scripts that may automate certain capabilities of the main application but do not fundamentally extend its functionality. This intermediate position — neither wholly autonomous nor functionally subservient — is expressed by the idea of *semi-autonomous* software agents.

Rigorous models of application networks among semi-autonomous components acquire an extra

[2] See https://pdfs.semanticscholar.org/1ad7/c459dc4f89f87719af1d7a6f30e6f58dff17.pdf; note that this is a different project than the ViSion Ontology referenced earlier.

level of complexity precisely because of this intermediate status: protocol definitions have to specify both the functional interdependence and the operational autonomy of different parts of the application network. We propose to address this complexity by adopting a hypergraph-based paradigm for procedural and type modeling, from which derives our proposed **HMCL** format.[3] The goal of **HMCL** is to define protocols for inter-application messaging by focusing on procedures enacted by both applications before and after each stage in the communication. This is not (in general) a "Remote Method Invocation" or "Simple Object Access Protocol" where one application explicitly initiates a specific procedure for the second to perform; instead, the chain of procedure calls (which we call the multi-application *operational semantics*) is more indirect, with *requests* and *responses* that may be mapped to different procedure-sequences in different contexts. Nevertheless, although one application does not need detailed knowledge of the other's internal procedure signatures (which would break encapsulation), a rigorous messaging protocol can be developed by specifying requirements on the relevant procedures implemented by each application. Developers can then explicitly state that a given set of procedures implements a given **HMCL** protocol (the multi-application documentation thereby has more detailed information about application-specific procedures than each application has vis-à-vis its peers). The functional interdependence between applications can accordingly be modeled by defining protocols which must be satisfied by procedure-sets internal to each end-point — the relevant information from an integrative standpoint is not the actual procedures involved, but confirmation that the relevant procedure sets adhere to the relevant multi-procedural protocol.

Procedural modeling is not only significant in the context of two or more separate applications sharing data; for some dimensions of software implementation even apart from multi-application networking, it is essential (or at least expedient) to employ rigorous procedural descriptions. One case in point is testing, particular integration testing and **GUI** unit testing — procedural models help programmers both design and implement tests. Indeed, testing in a **GUI** context is more complex then testing libraries whose behavior can be analyzed via command-line programs; with **GUI**s it is necessary to simulate or manually perform user actions for which a test entity seeks to demonstrate proper application response. For **GUI**s, then, testing requires either special-purpose versions of the application User Interface or a protocol wherein an external test suite communicates with the main application (or some combination of the two). In many cases, then, the connection between a test suite and its target applications serves as a de facto multi-application network, so that workflow models can also b e employed as testing engines (Figure 7 shows an example of a test suite which interacts with the data-set application showed in earlier screenshots via **TCP** networking).

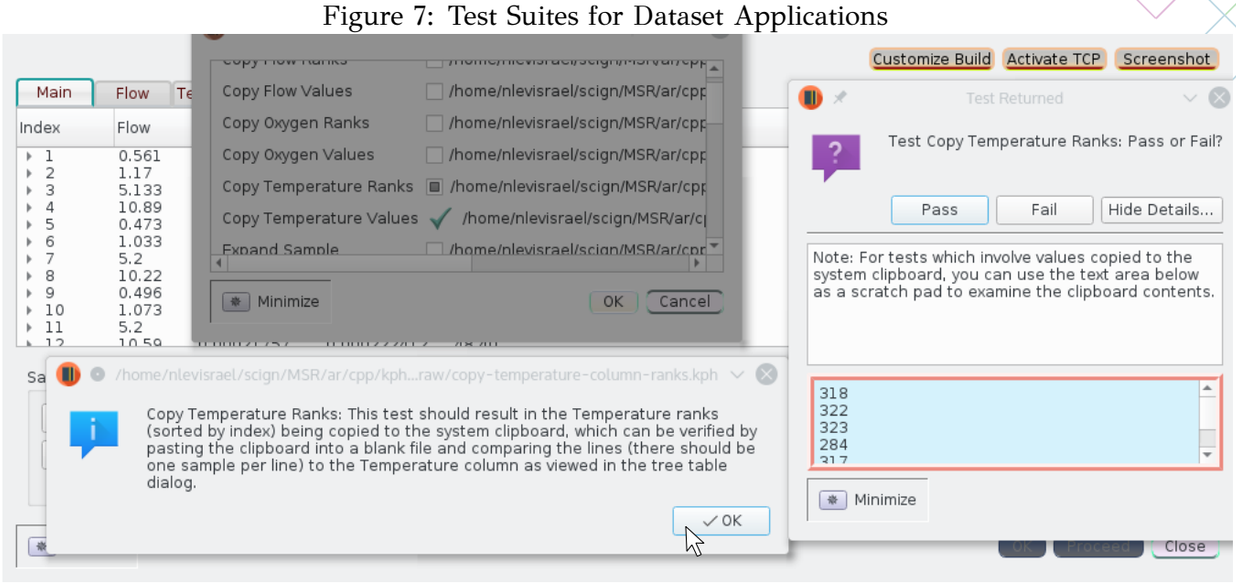
Furthermore, as discussed at the end of Part II, multi-application networking also overlaps with *heterogeneous database engineering* insofar as multiple applications (or multiple semi-autonomous agents) need to share a common data store. This means that information in the data space is characterized in part by procedural protocols guiding the use of particular aggregate data units. The **THQL** format, for representing hypergraph data in a database context, is designed in part to allow these protocol specifications to be encoded directly in the database.

Native/Desktop Application Development with THQL

As described earlier, **THQL** is a database engineering protocol which prioritizes data-persistence components that can be included in source code fashion within application-development projects. **THQL** is "transparent" in that all layers of data persistence and query processing logic are provided via self-contained source-code libraries. The complete database functionality can then be

[3] A preliminary characterization of hypergraph-based type systems (and their corresponding representation of detailed procedural signatures and requirements) can be found in Nathaniel Christen's chapter (chapter 3) in *Advances in Ubiquitous Computing: Cyber-Physical Systems, Smart Cities and Ecological Monitoring*, edited by Dr. Neustein (see https://www.elsevier.com/books/advances-in-ubiquitous-computing/neustein/978-0-12-816801-1).

Figure 7: Test Suites for Dataset Applications



statically examined via the source-code files, and dynamically examined by running the client application through a debugger. Moreover, because all THQL source code is bundled with application code, THQL can be configured to integrate seamlessly into its environing client-application logic. For example, THQL can be extended to natively recognize client-specific datatypes as data fields, or to execute client algorithms as query parameters.

As a query *language*, THQL can be instantiated either by special languages with their own syntax and semantics (analogous to SQL or SPARQL), or as an interface and pattern specified for a conventional language, such as C++. In the latter guise, THQL provides a common protocol for essential database tasks, such as constructing, updating, querying, and backing up database instances. Each procedure comprising the THQL protocol is assigned a specific role, so that the protocol can be abstractly modeled as a set of data-management roles mapped to corresponding procedure implementations. On this basis, custom query languages can be constructed by exposing each role-procedure to a scripting interface. For example, THQL's Reference Implementation (DigammaDB) exposes the protocol-specific procedures via a set of pointers to C++ functions. Consequently, parsing a query language is then rendered a straightforward process of mapping query expressions to the requisite procedures, whose corresponding handle can then be obtained via C++ interop.

As suggested by its name, THQL is centered around the operations to define and store hypergraph-form data: information which has several levels or scales of structuration. This means that the THQL protocol includes procedures for registering individual data fields (representing, in general, primitive types such as integers and decimals) in a database; aggregating fields into "hypernodes," or groups of interrelated information; connecting pairs of hypernodes by identifying a specific connection which they have; adding contextual details or annotations (via so-called *frames* and *channels*) which refine assertions of hypernode connections; and constructing "proxies" to database elements (e.g. hypernodes, frames, and subgraphs) which can be referenced (via unique identifiers) as individual data fields. Since proxies can then be aggregated into hypernodes in turn, THQL graphs can have, if desired, arbitrarily deep nested structures.

THQL also recognizes additional structures corresponding to conceptual details described earlier — for example, fields within hypernodes can be linked with dimensional attributes (e.g. scales and units of measurement) and identified as micro-citation targets. THQL likewise supports a genre of controlled vocabularies applying to hypernode-types and/or connection labels (called "dominions," for "Domain-Specific Mini-Ontologies"). Consequently, a graph can then be, if desired, configured to only accept hypernodes which conform to one of the dominion-defined types; and/or to only allow connections to be asserted between hypernodes when these connections

can be labeled from a dominion-specific list of connectors. Less restrictively, graphs can be defined in a more free-form style but use dominions to filter or query nodes and edges.

Another feature of THQL, relevant to application integration, is the notion of configuring each database to support different "modes" of data persistence. It is possible to use THQL for completely in-memory data management, with no direct data persistence at all. This would be an appropriate solution when data can be read all at once from a static source, such as a data set. In this guise, THQL would be used to build a structural model of the data set, which can then be queried by application code. Conversely, it is possible to employ THQL as a continuously-updated data store, where changes to an underlying THQL graph are persisted to disk as soon as they are registered. Between these extremes, THQL graphs can hold dynamically changing representations of a persistent database, which are only incorporated into the underlying database when instructed by the client application. To support these different operational modes, THQL engines need the capability to represent each data type in several different formats, tailored to different stages of processing through which values are routed before they can be stored persistently.

In DigammaDB (the THQL Reference Implementation), persistent data storage is implemented via the WhiteDB database engine. WhiteDB is a hybrid graph/record database which allocates a persistent data store in shared memory (allowing each database to be accessed from multiple applications). ϚDB encodes hypernodes in WhiteDB records (although programmers can interface with the underlying WhiteDB instances if desired). ϚDB can then use WhiteDB's index and query mechanism as the foundation for its own higher-level query system. ϚDB provides a convenient interface for binary-encoding user-defined C++ types, so that arbitrary application-level data can be stored via THQL (we can supply more documentation describing hypergraph-encoding with WhiteDB if needed). In addition — as an alternative to writing serializers for bonafide C++ types — it is possible to construct "ghost" types, which are Qt data structures built via the "QVariant" class, so that all (or most) hypernodes in the corresponding database have a single C++ type — this technique is appropriate when, for example, the purpose of a THQL instance is to read and then update HGXF files with new information. To support different THQL operational modes, ϚDB organizes a stage-structure based on encoding WhiteDB values: at the ground level, values are simply pointers to in-memory C++ objects. At an intermediate level, values are encoded (via the QDataStream class in Qt) into structures which recognize the WhiteDB encoding scheme but do not themselves interact with WhiteDB. Finally, values may be recorded as WhiteDB fields and records ready for persistent storage.

WhiteDB also allows databases to be shared (including being sent over a network) by storing all database information in a special file format. ϚDB instances can be shared via this same mechanism, although another option is to export the contents of a ϚDB database to HGXF files, which in turn form the core of a research data set representing the database contents at a specific moment in time. In this guise ϚDB works in conjunction with dsC, serving as the engine to construct a data set through which research data curated via a DigammaDB database can be published (dsC will be described in a later section). ϚDB also demonstrates how hypergraph databases — as well as data generated from these databases for data-sharing initiatives — can store data constrained by "hypergraph ontologies," which are discussed in the next section.

Hypergraph Ontologies

The notion of *hypergraph* ontologies extends the idea of *ontology* as this appears in the context of the Resource Description Framework (RDF). In the Semantic Web, an "ontology" is essentially a graph schema, defining metadata for any information that can be serialized or represented in graph-like fashion, as well as criteria which graphs must satisfy when they are used to encode a specific sort of data. To encode information about a *clinical trial*, for instance, one may require that graphs include one node (representing the trial itself) which is linked to other nodes representing

patients enrolled in the trial, as well as one or two nodes representing the trial's start and end dates. These requirements constitute both a structural mandate on the graphs — specifying how the nodes should be connected — and a *semantic* requirement on the nodes, each of which must be tagged with metadata clarifying that the corresponding node embodies a trial, person, or date. An ontology then supplies a fixed vocabulary with which to rigorously declare this necessary metadata and the relevant graph-construction rules. Insofar as graphs are employed to encode data, ontologies are analogous to Document Type Declarations (DTDs) in the context of XML.

As explained above, the goal of HGXF is to provide a general-purpose hypergraph representation format, suitable for all hypergraph data structures as well as any structures which are categorically subsumed by hypergraphs (such as RDF graphs). Consequently, HGXF is designed in part by examining existing Hypergraph Database software (such as HyperGraphDB, AllegroGraph, and Graken.ai) and runtime hypergraph libraries, as well as academic literature where hypergraph analyses have been used for (e.g.) image-segmentation and Machine Learning algorithms. These resources provide an overview of the range of data structures which need to be encoded by a general-purpose language such as HGXF. The design of HGXF has also been influenced by the Semantic Web, insofar as hypergraphs are a generalization of the directed, labeled graphs that are the building-blocks of the Semantic Web. At the same time, hypergraphs — along with structures that can be modeled via hypergraphs, such as Conceptual Spaces — are an improvement on Semantic Web data formats and schema, addressing the limitations of a paradigm devoted to modeling complex information via first-order logic and non-nested graphs, with no notion of scoping or locality.[4]

At the same time, a lot of effort has been extended building technologies to integrate heterogeneous data spaces via the Resource Description Framework (RDF) and RDF ontologies; it would be irresponsible to ignore these contributions. In radiology, for example, attempts to better incorporate clinical and outcomes data are centered on ontologies such as RadLex, VISION, and SEDI; insofar as these are (or have potential to become) canonical reporting standards, Patient-Centered research should promote rather than critique these initiatives. As a result, the important consideration is how to employ hypergraphs as an extension to the Semantic Web when warranted while preserving the virtues (and interoperating with) conventional RDF ontologies.

The idea that hypergraphs *extend* but do not *replace* RDF and the Semantic Web implies that hypergraph schema are extensions of (but not substitutes for) RDF ontologies — which in turn yields the notion of *hypergraph ontologies*. In a conventional RDF ontology, metadata is primarily associated with graph nodes and edges. In particular, nodes are referenced to Uniform Reference Identifiers (URIs), such as web addresses, and edges are labeled with concepts formally defined in one or more ontologies. Concepts which are used to annotate graph-edges, and which are given a fixed meaning in some controlled vocabulary, are often called "properties." One special "is-a" property is often used to connect nodes with concepts that classify entities into one of many categories defined in an ontology, often called "classes." As such, most RDF ontologies are primarily composed of *classes* and *properties*, each assigned a unique label. The purpose of metadata for a given graph is then to link nodes to classes (for example, specifying that one node represents a clinical trial and the second represents a patient), and furthermore link edges to properties (for example, specifying that a patient-node is connected to a trial-node in that the patient is *enrolled in* the trial).

Hypergraph ontologies are similar to conventional RDF ontologies in that they likewise provide constraints and metadata for graphs. However, hypergraph ontologies are more complex because hypergraphs are likewise more complex than ordinary graphs. In particular, hypergraphs have

[4] These are familiar critiques, but laid out with particular thoroughness by the Conceptual Space community: see http://idwebhost-202-147.ethz.ch/Publications/RefConferences/ICSC_2009_AdamsRaubal_Camera-FINAL.pdf and https://pdfs.semanticscholar.org/1d58/faa5cb23efb7394aeb3dfe688edd99797a91.pdf.

different layers of structure: whereas RDF nodes are intended to represent a single concept or value (such as a number, date, personal name, or URL), a *hypernode*, within a graph, typically encompasses multiple pieces of information inside it (often called *hyponodes*, *projections*, *inner elements*, *roles*, or just *nodes*).[5] In general, when analyzing hypergraphs it is necessary to distinguish at least two "tiers" of nodes, *hypernodes* and *hyponodes*, such that hyponodes are contained within hypernodes. As a result, hypergraph ontologies need a corresponding distinction for node and edge annotations: insofar as nodes are categorized via classes, and edges via properties, it is necessary to stipulate whether these classifications apply to hypernodes, hyponodes, or some combination of the two.

A further complication (contributing to the complexity of hypergraph ontologies compared to RDF ontologies) arises because, even though hypergraphs represent nested or hierarchical structures, these hierarchies are often partial or overlapping. For example, a patient is *part of* a clinical trial, but a patient is also included in other collections as well; for instance, a patient may be enrolled in a specific health plan (for insurance coverage). One technique for modeling overlapping hierarchical data via hypergraphs is to employ "proxies," which are digital identifiers encoding a multi-faceted concept into a single value that can be part of a hypernode (proxies are similar to "foreign keys" in SQL). Therefore, each patient, represented by its own hypernode, has an identifier which can be a proxy-value for the patient; for example, a value assigned to a hyponode becomes included in the hypernode encoding the list of patients enrolled in a clinical trial, or in the hypernode encoding the list of patients enrolled in a specific health plan. Hypernodes can then be linked to other hypernodes by virtue of proxies (e.g., the trial-to-patient connection), and also by virtue of overlap (e.g., the set of all patients both enrolled in a given clinical trial *and* enrolled in a given health plan).

In sum, compared to RDF — where there is one single sort of node-to-node relationship, based on whether or not an edge exists between nodes and how this edge is labeled — hypergraphs are more flexible/expressive because they have multiple genres of node-to-node relationships: the relation between hypernodes and their inner hyponodes; between hypernodes and one another; between hyponodes in different hypernodes; and variations on each of these relation-types wherein relations are defined indirectly through proxies. Moreover, in addition to hypernodes and hyponodes, hypergraphs afford additional levels of detail, such as *frames*, *channels*, and *axiations*. All of these details provide different "sites" where hypergraph annotations and metadata may be defined.[6]

An additional distinction within the Semantic Web is the contrast between *reference ontologies* and *application ontologies*. In general, *reference ontologies* are general-purpose schema intended to establish conventions shared by many different applications, to ensure that a large collection of data-producing software in a given domain is interoperable. By contrast, *application ontologies* are narrower in scope because they are more tightly integrated into applications that directly send and receive data. Ontologies of either variety are used by software to interoperate with other software: so long as two applications are using the same ontologies, it is possible to ensure that one application can understand the data produced by a second, and vice-versa. However, such inter-operability is only potential; it is the responsibility of programmers to actually implement code which produces and/or consumes data that conforms to the relevant ontology specifications. In general, application ontologies are structured in such a way that these concrete implementations are more straightforward to produce, compared with reference ontologies. Reference ontologies

---

[5] The term "roles" is used by Grakn.ai (see https://dev.grakn.ai/docs/schema/concepts); "projections" is used by HyperGraphDB (see http://www.hypergraphdb.org/?project=hypergraphdbpage=RefCustomTypes); "inner entity" is used by the biointelligence project (see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3555311/), where the corresponding notion of "external entity" refers to what in other contexts might be called other hypernodes linked to a given hypernode via hyperedges.

[6] This means that formats for describing hypergraph ontologies have to be more expressive than RDF ontologies, because RDF ontologies need only to classify metadata as node-annotations or edge-annotations; by contrast, hypergraph ontologies need to distribute annotations among multiple sites of graph structure.

offer little guidance to developers vis-à-vis how to directly support the ontology within application code. Conversely, application ontologies more rigorously describe the data structures which applications must implement in order to properly manipulate data that is structured according to the specifications of the ontology.

Within data mining and image analysis, hypergraph models are used in different ways for different algorithms. In the context of Covid-19 radiology, https://arxiv.org/pdf/2005.04043.pdf describes an algorithm for assessing the probability of SARS-CoV-2 infection from chest **CT** scans, where hypernodes represent high-dimensional vectors (191 dimensions overall) and hyperedges represent k-nearest-neighbors; here each hypernode represents an entire image, mapped to a 191-dimensional feature-vector. In contrast, other image-analysis methods use hypernodes to designate smaller segments *within* the image, where hyperedges designate geometric adjacency and/or feature-space similarities. Whatever the algorithm, hypergraph analyses would employ a hypergraph library to store preliminary data for analysis and/or for serialization within a data set. One benefit of a Hypergraph Application Ontology is therefore that these data structures used internally to implement analytic methods can be directly expressed within the ontology, whereas **RDF** ontologies can only model hypergraphs indirectly.

Although it is theoretically possible to encode data directly via **RDF** graphs, it is far more common for applications to employ tabular and/or hierarchical formats such as spreadsheets, Protocol Buffers, **XML**, or **JSON**. As a result, the role of ontologies for constraining data structures (so that they adhere to common standards) is indirect. It is useful to remember that ontologies are, at their most basic level, Controlled Vocabularies; as such, ontology constraints often amount to stipulating a set of acceptable terms for a data value, column header, or annotation. For a trivial example, our calendar recognizes 12 month names and 7 day names, which constrain the set of values permissible for "month" and "day" within a calendar date. These terms are so commonplace that a "date ontology" is unnecessary, but in scientific or technical domains it becomes necessary to define vocabularies of allowable names or labels for specific data fields that representing some scientific value or measurement. For instance, the Ontology of Vaccine Adverse Events (see https://jbiomedsem.biomedcentral.com/articles/10.1186/2041-1480-4-40) provides a nomenclature for use in Adverse Events Reporting, so that researchers or clinicians can describe symptoms via canonically recognized terms rather than through informal text descriptions. In general, ontologies constrain data sets by stipulating that particular individual values within the overall data collection have names or descriptions whose associated set of possible values is prescribed *a priori* by the applicable ontology. However, the relationship between ontologies and concrete data sets must itself be documented, which is where application ontologies can become relevant — application ontologies provide a bridge between reference ontologies and the applications which use them (along with the data generated and shared by those applications).

In order to preserve the benefits of **RDF** ontologies — while also addressing those lacunae which make the Semantic Web "not (really) semantic" — hypergraph ontologies need to model constraint schema on hypergraph constructions (which have significantly more parameters of structuration than **RDF** graphs) while also connecting these schemas to the Controlled Vocabularies and logical axioms of Semantic Web (particularly **OWL**) ontologies. There are as such several areas of detail within hypergraphs where links to **RDF** ontologies may be drawn, which are outlined here:[7]

Hypernodes' Cocyclic Type Structure One of the central principles of hypergraph data modeling is the use of *hypernode types* to specify what sort of information is necessarily associated with a hypernode. In particular, a hypernode encompasses multiple hyponodes, each with their own type. These hyponodes represent information in some sense "contained within" or

[7] A full explanation of these concepts and terminology depends on an in-depth treatment of hypergraph type theory, which is outside the scope of this proposal.

"tightly connected to" a hypernode (whereas data less canonically associated with each hypernode would, in general, be asserted via hyperedges rather than via hypernode/hyponode containment. In order to ensure that hypergraphs are predictably organized, hypernodes cannot have arbitrary collections of hyponodes, but must instead be aggregates of hyponodes which are assembled according to a schematic pattern, defined in terms of hyponode types. For maximum generality, a hypergraph type system should allow hyponode-type patterns to be as flexible as possible without introducing a need for metadata asserted at the level of individual hypernodes rather than hypernode types; this motivates the idea of a "cocyclic" type system which is minimally constrained (but not unconstrained).[8] When translating RDF ontologies to hypergraph schema, then, one consideration is whether edge-requirements are sufficiently ubiquitous in some context (e.g. with respect to some **rdf:class**) that these edges should be translated to hypernode/hyponode inclusions, and then to define a pattern of hyponode types for the corresponding hypernode type.

Roles, Projections, and Dimensional Annotations A hypernode type provides a schema defining a sequence of hyponode types; it is sometimes said that the hypernode "projects onto" that space of hyponode types. This projection is minimally characterized by hyponode types, but some hypergraph systems allow the projection to be *annotated*, introducing additional metadata that constrain (or augment the expressive power of) the enclosing hypergraph. Annotations can define scales/units/levels of measurement, probability distributions, situational roles, and other details lending semantic grounding to the data-field encapsulated by a hyponode. This metadata can then be a vehicle for translating RDF class constraints to hypergraph schema.

Semantic Nominal Dimensions The most direct translation of Controlled Vocabularies to a hypergraph context is often that of constraining the space of variation for one specific project to a set of allowable terms. In the typical case, a hyponode type encapsulates a nominal set of values (i.e., an enumeration), so any hypernode including that type as one of its projects is constrained by the labels registered in the vocabulary (a related formulation replaces non-hierarchical vocabularies with *taxonomies*, where some labels are treated as more or less granular variants of others).

Dimension Aggregates, Domains, and Conceptual Spaces Conceptual Space Theory — which has been applied toward the analysis of data structures and cognitive patterns in science, linguistics, and AI — can be modeled in the hypergraph context by noting that hyponode projections are sometimes interdependent: dimensions tend to aggregate into semantically related groups (like *latitude* and *longitude* as geographic markers). In Conceptual Space models, accordingly, projections are split into two levels — *dimensions* and *domains* — while other dimensional-analytic constructions (such as scales and units of measurement) are carried over.[9] Conceptual Space Theory also then introduces concepts of fuzzy logic or "convexivity" (according to different metrics) to simulate patterns in human conceptualization.[10]

Probabilistic, Temporal, and Overlapping Hypergraphs Other forms of metadata constrained via ontologies can be expressed in terms of annotations defining weights or probabilities on hypernodes and/or hyperedges. One example is the juxtaposition of alternative markup hierarchies, in the context of hypergraph representations for Concurrent Markup languages such as . Numeric edge-annotation can represent weights (e.g., provide measures of the degree of uncer-

---

[8] A pattern of hyponode types can be called "cocyclic" if the type-sequence includes a (possibly empty) tuple of types with no requisite pattern (called the "precycle") followed by a repeatable type-tuple. Cocyclically typed hypernodes therefore represent expandable data structures such as lists, stacks, queues, deques, and dictionaries. A typical hypernode type may indirectly include multiple collections-types via proxies.

[9] See https://pdfs.semanticscholar.org/521a/cbab9658df27acd9f40bba2b9445f75d681c.pdf, https://arxiv.org/pdf/1703.08314.pdf, or http://lup.lub.lu.se/record/1775234.

[10] A good review is provided by the publications and code archived at https://github.com/lbechberger/ConceptualSpaces.

tainty in the edge's relation actually obtaining), but constructions similar to weights have other sorts of applications. For instance, edge-annotations can be measures of time-spans, allowing hypergraphs to describe "entity-event models."[11] These models are particularly inportant for patient-centered data via their use in analyzing clinical outcomes by aggregating data according to patient-centered reviews extracted from data.[12]

Proxies, Inverted Proxies, and Double-Inverted-Proxy Constructions As described earlier, hypernodes can assert "containment" of other hypernodes by containing a *hyponode* which *proxies* the second hypernode. An *inverted proxy* connection is therefore the mirror-image of this assertion (which may or may not be formally recognized by the hypergraph). A *double-inverted-proxy* connection is accordingly the relation obtaining between two hypernodes which are both proxied by one third hypernode (using the earlier example of proxies, the fact that two patients are enrolled in the same clinical trial). Many graph connections identified in a Semantic Web context (e.g., by **SPARQL** queries) are likely to translated to double-inverted proxies in a hypergraph context.

In general, these hypergraph constructions represent sites for asserting constraints that (for **RDF** ontologies) would be defined on classes or properties; they are therefore a natural scaffolding for translating **RDF** ontologies to hypergraphs. Such a translation mechanism allows existing ontologies — which may play a valuable role in specifying protocols for workflows and data-sharing between software components — to be reused in a hypergraph modeling environment.

As illustrated by **CAPTK**, multi-application workflows are characterized both by the data which is transferred between applications and by the operations which connect the two applications — that is, the procedures enacted by each application when they become operationally linked. As a preliminary model, we can identify two stages of operational connection between an already-running application (which may be called the *primary* component) and a second application launched by the primary (which may be called the *peer* component). The first stage occurs when the primary component launches the peer component, and is characterized by two operational sequences: procedures enacted by the primary prior to this launch, and procedures enacted by the peer subsequent to the launch. A second stage occurs when the peer component has completed its actions, and sends data back to the primary component, which again involves two operational sequences: procedures enacted by the peer prior to the transfer, and procedures enacted by the primary subsequent to the transfer. Fully describing the procedural workflow therefore entails specifying four operational sequences: primary, then peer, during the launch stage; and peer, then primary, during the transfer stage. A schema describing the operations performed during these four sequences can be called a *procedural ontology*.

Consequently, rigorous models of multi-application networks should *synthesize* information about data structures (the type of information shared between application-points) with information about procedural workflows (describing operational sequences prior to the launch and transfer stages of a multi-application linkage). The synthesis of this structural and procedural information can be called a *procedural application ontology*. Insofar as the new **HMCL** language defines the operational semantics of multi-application workflows, **HMCL** can be seen as a format for asserting and integrating procedural application ontologies (according to this definition).

Dataset Creator

Dataset Creator ($_{ds}$C) is a framework for constructing data sets which include computer code based on the **QT** application-development platform. Dataset Creator takes advantage of the **QT**

[11] See https://allegrograph.com/consulting/entity-event-knowledge-graphs/.
[12] See http://exploreclg.montefiore.org/upload/training-materials/The%20Cohort%20ParadigmV30.pdf.

platform to construct Research Objects with exceptional GUI and data-mining capabilities. QT, the leading native cross-platform development toolkit, is a comprehensive framework encompassing a thorough inventory of programming features — networking, GUI implementation, file management, data visualization, 3D graphics, and so forth. Data sets based on QT require users to obtain a copy of the QT platform, but QT is free for non-commercial use and easy to install — importantly, QT is wholly contained in its own folder and does not affect any other files on the user's computers (in this manner QT is different than most software packages, which usually demand a "system install").

By leveraging the QT platform, dsC enables standalone, self-contained, and full-featured native/desktop applications to be uniquely implemented for each data set, distributed in source-code fashion along with raw research data. Adopting such a data-curation method makes data sets easier to use across a wide range of scientific disciplines, because the data sets are freed from having to rely on domain-specific software (software which may be commonly used in one scientific field but is unfamiliar outside that field). In addition, Research Objects composed with dsC can be integrated into Multi-Application Networks (which are described above) because the dataset applications are autonomous native GUI applications that can easily interoperate via QT messaging protocols.

Because every data set is unique, each Dataset Application will necessarily include some code specific to that one Research Object. However, dsC will provide a core code base and file layout which is shared by all dsC data sets by default. This common core is structured in part by the goal of developing Dataset Applications in a QT context; for instance, dsC projects are structured to use QT's "qmake" build system as the primary tool for compiling data-set code. The common dsC code therefore includes qmake project files which support compiling application with several build configurations. In this framework, data-set users are classified into several different roles — in addition to ordinary users (specifically, researchers who want to work with and draw information from data sets but have no development connection to these data sets themselves), dsC recognizes roles for authors, editors, testers, and other users who are responsible for bringing data sets into publication-ready form to begin with. Depending on the administrative role, data set code can be compiled with additional features (e.g., unit testing features).

Another core component of dsC is LaTeX code that authors may use when preparing documents accompanying their data set. These LaTeX files encompass special functionality for defining code annotations and semantically significant points in article text, such as sentence and paragraph boundaries. This LaTeX code can be used in conjunction with a pre-processor that generates LaTeX files from a special input language. The goal of these text-processing technologies is to improve the interoperability between research papers and data sets. In particular, the LaTeX pre-processors (and subsequent LaTeX-to-PDF converters) generate HGXF files which store information about textual and PDF viewport coordinates specifying the location of semantically meaningful elements such as sentences and annotations. These files are then zipped and embedded in PDF files. With dsC, the resulting PDF files can then be loaded into a customized PDF viewer capable of reading the embedded HGXF data. This allows the PDF application to utilize the embedded information so as to provide a more interactive reading experience — for instance, viewing annotations or copying sentences via context menus, where viewport data maps cursor position to textual elements visible on the PDF page. These features provide an application-level interface between the PDF viewers (considered as GUI components) and the corresponding GUI components in Dataset Applications.

With proper customization, both the PDF viewers and the dsC Dataset Applications can interoperate, with PDF context menus calling up windows in the Dataset Application's GUI implementation, and vice-versa. For example, researchers reading the PDF version of a scientific article can launch the Dataset Application to explore some detail mentioned in the text. This is an example of where micro-citations are practically useful: any microcitable element in a document (such as

a table, column, row/record, or analytic procedure formalized as a procedural asset associated with a data set) can be linked to a corresponding GUI element in the Dataset Application. For example, a statistical parameter — mentioned by name in the text, and perhaps represented in serialization within raw data — can be mapped to a GUI table column, and specifically the column header; this is then an annotation target, in the sense that for readers to gain more information it is proper to link mentions of the relevant scientific concept in article text to the column header as a graphical element that can be made visible. The link is operationalized by implementing a procedure to show the GUI window where the table is located, and ensure that the column header lies in the visible portion of the screen, as a response to readers on the PDF side signaling a desire for information on the annotated text element. In the opposite direction, database elements can be annotated with links that can used by the Dataset Application to launch a PDF window opened to the page and location where the corresponding concept is discussed in the article.

In order to properly model this semantic, viewport, and data set data integration, dsC uses a new document-representation format called HTXN (Hypergraph Text Encoding Protocol). With dsC, HTXN files are not only associated with data set assets; they are also machine-readable document encodings that can be introduced into publication repositories and other corpora oriented toward machine-readability. Authors can host HTXN files within data sets and link to them via services such as CrossRef, thereby ensuring that a highly structured, machine-readable version of their papers is available for text and data mining. The HTXN protocol is also useful for encoding natural-language content which becomes part of a data set as data assets in themselves; for example, patient narratives.

Further documentation of text-encoding methodology (applicable to both patient narratives and publications associated with CR2 research data) is available on the CR2 web site: such as here (this is a downloadable PDF link; visit the repository to see the larger archive structure). The document referenced in this hyperlink contains more information on dsC, HGXF, HTXN, and the other technologies discussed here.

Part IV: Case-Studies in Biomedical Data Integration

Integrating Clinical Outcomes with Radiological Data

Diagnostic imaging and radiological data sets are an important aspect of Covid-19 data because radiological analyses of the chest, particularly Computed Tomography scans, can prove the suspicion that a patient is suffering from Covid-19. The Radiological Society of North America (RSNA) has announced an initiative to create an "AI Imaging Data Repository," which will "support research on using medical imaging to screen for, diagnose and treat COVID-19" (see https://www.rsna.org/covid-19). In accordance with previously curated RSNA data sets, this repository will include image series paired with annotations encoding analytic results indicating image features that suggest a Covid-19 diagnosis.

To demonstrate the need for radiology-oriented ontologies designed with an emphasis on application integration — that is, for an Application Ontology dedicated to the diagnostic-imaging domain — one should note that curating radiological data sets presents challenges different than those addressed by conventional diagnostic-imaging software. In the conventional workflow, radiographic images are requested by some medical institution for diagnostic purposes. Relevant information is therefore shared between two end-points: the institution which prescribes a diagostic evaluation and the radiologist or laboratory which analyzes the resulting images. Building radiographic data repositories complicates this workflow because a third entity becomes involved — the organization responsible for aggregating images and analyses is generally distinct from both the prescribing institution and the radiologists themselves. Large-scale repositories (such as the RSNA's AI Imaging Data Repository) are designed to pool patient data from multiple hospitals

and multiple radiographic laboratories. As a result, both radiologists and prescribing institutions, upon participation in the formation of the target repository, must identify when a given patient is a proper candidate for the repository. For example, images become relevant for the Covid-19 AI Imaging Data Repository when such images consist of CT scans that are prescribed for the purpose of confirming a diagnosis of SARS-CoV-2 infection.

In the diagnostic context, radiographic data may, in general, include the following: (1) the findings reported by radiologists; (2) image-annotations made by radiologists which point to image features providing evidence for their findings; and (3) data generated by algorithmic image-analysis tools. As AI tools and Computer Vision have become more powerful, data generated by quantitative image-analysis has become an important element of radiographic data. Moreover, the software used to perform image analysis may be distinct from the PACS systems employed by radiologists themselves in the laboratory setting. For example, the CaPTk (Cancer Imaging Phenomics Toolkit) software, developed by CBICA (the Center for Biomedical Image Computing and Analytics at the University of Pennsylvania), is designed as a central workstation from which — after loading an image — researchers can initiate a number of analytic operations, yielding a modified image and/or a quantitative data aggregate (e.g., a feature vector) encoding information about the image (see https://www.med.upenn.edu/cbica/captk/).

The CaPTk architecture, although developed in the context of cancer research, is compatible with image-processing for any radiological area of study. This architecture is extensible: developers may add new image-analysis software which is registered with the central CaPTk application, allowing users to launch these software extensions while running the main application. Collectively, then, a CaPTk instance enhanced with such software add-ons form a multi-application network similar to those outlined above in the White Paper. For this system to work properly, CaPTk and its peer applications implement a specific data-sharing protocol, which in turn is based on the Common Workflow Language (CWL) and on technical Qt coding patterns related to reactive programming.

In the context of radiological data curation — that is, where image-analysis data is not only used for a single diagnosis, but is also registered with a central repository which tracks radiographic data specific to a given disease, imaging modality, or course of treatment — the native data-sharing protocol between CaPTk and its peer applications can then be extended. In this manner, image-analysis data can be exported outside the overall CaPTk system, allowing this data to be shared with external repositories. Furthermore, a similar workflow can be implemented in the context of conventional radiological software — that is, software oriented to manual diagnosis and annotation rather than AI tools. For example, CaPTk pairs conveniently with MedInria (a general-purpose radiology platform) and with 3DimViewer (a tool which generates 3D tissue models from 2D image series). All three of these applications share a common programming foundation (based on C++, Qt, and the Insight Toolkit and Visualization Toolkit for image segmentation and 2D/3D image analysis). It is therefore possible to implement a data-sharing protocol which integrates each of these applications; a researcher could use MedInria for manual examination of an image series and then switch to CaPTk to perform AI-driven analyses. This data integration applies not only to radiologists conducting a prescribed diagnosis, but also — perhaps more significantly — to researchers conducting analyses on images published within a radiographic repository. In this case, researchers are trying to retroactively analyze collections of images, drawn from different patients, so as to find correlations between observable features in a diagnostic image and post-diagnostic treatment outcomes.

Integrating CaPTk (along with its software extensions, which are themselves semi-autonomous native applications) with additional radiological software, such as MedInria and 3DimViewer, yields a multi-faceted Application Network capable of integrating different forms of radiographic data. Such a framework, however, requires a data-sharing protocol which operates effectively at the ap-

plication level — in effect, an Application Ontology. Although the Common Workflow Language (CWL) provides a basic layer of inter-application communication, a fully-developed radiological application ontology has not yet been formalized. In curating the CR2 and AIM-Concepts repositories, we will address these lacunae by defining a diagnostic-imaging Hypergraph Ontology and implementing support for the corresponding schema as extensions to CaPTk, medInria, 3DimViewer, and other image-processing and research tools. We will also expand the scope of this ontology to incoporate clinical and outcomes data.

As illustrated by CaPTk, multi-application workflows are characterized both by the data which is transferred between applications and by the operations which connect the two applications — that is, the procedures enacted by each application when they become operationally linked. As a preliminary model, we can identify two stages of operational connection between an already-running application (which may be called the *primary* component) and a second application launched by the primary (which may be called the *peer* component). The first stage occurs when the primary component launches the peer component, and is characterized by two operational sequences: procedures enacted by the primary prior to this launch, and procedures enacted by the peer subsequent to the launch. A second stage occurs when the peer component has completed its actions, and sends data back to the primary component, which again involves two operational sequences: procedures enacted by the peer prior to the transfer, and procedures enacted by the primary subsequent to the transfer. Fully describing the procedural workflow therefore entails specifying four operational sequences: primary, then peer, during the launch stage; and peer, then primary, during the transfer stage. A schema describing the operations performed during these four sequences can be called a *procedural ontology*.

Consequently, rigorous models of multi-application networks should *synthesize* information about data structures (the type of information shared between application-points) with information about procedural workflows (describing operational sequences prior to the launch and transfer stages of a multi-application linkage). The synthesis of this structural and procedural information can be called a *procedural application ontology*. In the case of CaPTk, such a procedural model is implicitly recognized in the protocol for integrating new applications with the CaPTk software, but this implicit protocol has not been explicitly defined in ontological terms. As such, we propose to standardize a Procedural Application Ontology model which can clarify the inner workings of multi-application protocols such as those of CaPTk.

Aside from enhancing CaPTk itself, this form of ontology can serve the more general purpose of prototyping the kinds of inter-application networking which is prerequisite for generalizing medical-imaging software in the direction of data-repository curation and Comparative Effectiveness research. That is, projects which seek to aggregate radiographic data into third-party archives — such as the RSNA AI repository for Covid-19 — and/or projects which perform analyses cross-referencing radiographic data with clinical outcomes, need to pull data from radiological software systems so that this information can be absorbed into a larger clinical-evaluative ecosystem. In effect, radiological application networks need to be expanded to accommodate patient-centered, epidemiological, treatment plan, and outcomes data. We propose to formalize this extension phenomenon by treating it as a generalization of multi-application protocols already evidenced by tools such as CaPTk and analyzable in terms of Procedural Application Ontologies.[13]

Radiological data repositories may be developed in part to improve image-analysis in general — certain image sets can provide a "standard corpus" with which to compare different analytic methods — but a key goal of these repositories is also to cross-reference image analysis with clinical data and medical outcomes. For example, researchers may wish to devise algorithms for

[13] It should be noted that other software platforms — such as Paraview, in the context of data visualization, and CERN ROOT, in the context of nuclear physics, have related extension mechanisms that can also be analyzed as further details for a general-purpose model of multi-native-application networking.

analyzing diagnostic images with which to guide subsequent clinical interventions: the ultimate goal is to automatically extract image data, given an image series (together with clinical information pertinent to the clinicians' decision to request radiological evaluation), which suggests that one or another treatment plan is more likely to produce positive outcomes for the patient.

The manner in which both radiological and clinical software are structured, however, often makes it difficult to cross-reference radiological data with corresponding details describing patients' treatment protocols and medical outcomes once a radiological diagnosis has been performed. For a concrete example (see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5119276/pdf/nihms-822829.pdf), "to retrieve the radiation treatment plan" associated with a **CT** scan, one "has to search for the RTSTRUCT object based on the specific CT scan, and from there search for the RTPLAN object based on the RTSTRUCT object [which] is an inefficient operation because all RTSTRUCT [and] RTPLAN files for the patient need to be processed to find the correct treatment plan."

Even within the context of analyzing radiological data sets for radiology-specific data, established **PACS** software is not designed for large-scale query evaluation linked to clinical data. As pointed out by the Semantic **DICOM** documentation (see https://semantic-dicom.com/starting-page/), queries like "display all patients with a bronchial carcinoma bigger than 50 cm^3" cannot be processed by **PACS** systems: "although there are various powerful clinical applications to process image data and image data series to create significant clinical analyses, none of these analytic results can be merged with the clinical data of a single patient. However, retrieving this data is an essential prerequisite to perform such search queries." These inefficiencies limit researchers' ability to perform data mining which incorporates diagnostic images, and also to prepare well-integrated data sets for advancing diagnostic and Comparative Effectiveness research.

In a well-integrated research framework, treatment plans should be linked both to the diagnostic data which form the basis of the plan (including radiological images) and to reports of treatment outcomes. As a concrete example, consider the choice to use **IORT** (Intraoperative Radiation Therapy) as an alternative to conventional cancer treatments (see https://www.cuimc.columbia.edu/news/new-cancer-treatment-personalized-radiation-therapy-during-cancer-surgery). Researchers can ask two different questions related to the effectiveness of **IORT**: (1) which diagnostic cues indicate that a given patient may benefit from **IORT** in lieu of longer-term therapies; and (2) which treatment is more effective at promoting better survival rates. These two lines of inquiry both require an integrated data-querying mechanism that recognizes data from image analysis, clinical outcomes, and descriptions of medical treatment; each of these kinds of data can be potentially included as a source of parameters with which to define patient cohorts or to search for statistical correlations. The relatively new **IORT** treatment is an example of why integrated research is important — oncologists still have limited understanding of which diagnostic clues are statistically associated with positive outcomes from **IORT** as compared with conventional therapy.

In the context of Covid-19, which is an even newer research priority, initiatives such as the **RSNA** repository are just beginning to aggregate radiological data in a manner which permits broad-scale analysis to be performed. Although radiological ontologies are primarily designed for use in laboratory software itself — i.e., the programs radiologists use directly to derive and present their findings — radiological research in general introduces further application domains, including software for image analysis, for integrating radiological and clinical data, and for examining radiological data sets. This new data presents opportunities for the use of semantic annotation technology, including ontologies such as **SeDI** or **ViSion** (see https://epos.myesr.org/esr/viewing/index.php?module=viewing_poster&task=&pi=155548).

Insofar as **PACS** and **DICOM** systems need to be paired with applications in the other categories (such as image and data-set analysis), it is important to develop radiological Application Ontologies, alongside the canonical Reference Ontologies associated with diagnostic imaging. As

such, starting with the Covid-19 context, but designed for more general applications as well, one component of **CR2** is a novel radiology-focused *application* ontology, specifically what LTS calls the "Hypergraph Ontology for Diagnostic Imaging, Clinical Outcomes, and Data Mining" (**h-DICOM**). The goal of **h-DICOM** is to connect **RDF**-based radiological ontologies with software applications designed to manipulate radiological (and corresponding treatment and outcomes) data. In addition, **h-DICOM** will formalize the data-sharing protocols implicitly adopted by software aggregates such as **CaPTk**.

Within data mining and image analysis, hypergraph models are used in different ways for different algorithms. In the context of Covid-19 radiology, https://arxiv.org/pdf/2005.04043.pdf describes an algorithm for assessing the probability of SARS-CoV-2 infection from chest **CT** scans, where hypernodes represent high-dimensional vectors (191 dimensions overall) and hyperedges represent k-nearest-neighbors; here each hypernode represents an entire image, mapped to a 191-dimensional feature-vector. In contrast, other image-analysis methods use hypernodes to designate smaller segments *within* the image, where hyperedges designate geometric adjacency and/or feature-space similarities. Whatever the algorithm, hypergraph analyses would employ a hypergraph library to store preliminary data for analysis and/or for serialization within a data set. One benefit of a Hypergraph Application Ontology is therefore that these data structures used internally to implement analytic methods can be directly expressed within the ontology, whereas **RDF** ontologies can only model hypergraphs indirectly.

Image Analysis and GIS Imaging in Vaccine and Immunology Research

As with radiological outcomes and Clinical Effectiveness Research, vaccine ontologies are multimodal because the scope of vaccine research spans numerous scientific disciplines, including fields where quantitative or image-based analysis is methodologically significant. As such, data generated during clinical trials is only one part of our overall understanding of the immunological properties of different vaccines. Vaccine data can be partitioned into information generated *before* a vaccine is proven effective (that is, data generated during clinical trials, including challenge trials) and information generated *after* a vaccine is adopted for immunization campaigns. Data in the clinical trial phase tends to fit the structure of patient cohorts, demographics, and outcomes, whereas data in the vaccine "deployment" phase can be more epidemiological/sociological by virtue of post-vaccination analysis identifying the effects on the population writ large. In particular, modeling immunization drives often depends on gathering geographical and political information — that is, researchers need to identify localities where vaccinations are performed and to be able to model the governmental/organizational initiatives which support immunization.

Aside from these geographic, epidemiological, and clinical data profiles, vaccine research also addresses the immunological mechanisms which determine how a particular vaccine works biologically, as well as the level of immunity which the vaccine provides for individual patients. Understanding these phenomena requires observing the immunological response of patients upon their receiving the vaccine, which generally involves blood and serum analysis — using a variety of modalities, including chemical, genomic, and molecular imaging. Consequently, a general-purpose vaccine and immunization software suite needs the capacity to represent a broad spectrum of data which may be presented as evidence of a vaccine's immunological effectiveness. Biomedical imaging at the cellular and molecular level has become an increasingly important part of vaccine research, insofar as new imaging techniques allow scientists to directly observe immunological responses at the scale of antigens, adjuvants, and immune cells: "monitoring vaccine components, such as antigen or adjuvants, and immune cell dynamics at the site of vaccination or draining lymph nodes can provide important information to understand more about the vaccine response." As such, "a variety of imaging modalities including bioluminescence imaging, nuclear medicine imaging (such as positron emission tomography [PET], single photon emission computed tomography [SPECT]), and magnetic resonance imaging (MRI) can provide in vivo non-invasive

imaging for visualizing immune cell kinetics" (see https://pubmed.ncbi.nlm.nih.gov/31406689/).

The evolution of vaccine research is, in effect, an inversion of that of diagnostic imaging. That is, in the context of radiology, image-based technology have become established in previous decades and have only quite recently been augmented with data-integration models that allow images to be cross-referenced with clinical and outcomes data. Conversely, in the vaccine context, the *clinical* and epidemiological/sociodemographic dimensions of vaccine effectiveness have been modeled for decades, but only recently has sophisticated medical imaging become part of the vaccine engineering arsenal. Despite these inverted paths, one can observe that contemporary radiological and vaccine research have a similar technological infrastructure, characterized by a digital synthesis of graphical/quantitative and clinical information.

Given the interdisciplinary nature of vaccine research, software engineers developing tools for curating data sets involving vaccine research should anticipate a similar diversity in the information present within repositories which include vaccine-related research. That is to say, data-set software relevant in the vaccinology context should be able to read data in *diverse formats*, which represent the scope of vaccine investigations (clinical trials, cytometry, blood/serum analysis, bioimaging, etc.). Similarly, developers should provide a suite of software tools offering interactive visualization of this data spectrum, including imaging tools. The tools for **CR2** will therefore include components to help developers create software supporting vaccine/immunology research in several distinct areas, two examples of which are reviewed here in greater detail.

A Pure-C++ Library for Cytometry

The **CR2** code libraries will include components to manipulate file and data types associated with cellular, biomolecular, and immunological imaging. Examination of cellular-scale processes has several different roles in vaccine/immunization research, including identification of viral proteins and observing the immune system response to viral infection and/or vaccination. Moreover, sudies of cellular processes can occur either by directly visualizing phenomena on this scale or by indirect analytic methods, such as cytometry. In "flow cytometry", for instance, a stream of cells is investigated by passing them through laser beams where the cells' material absorbs and/or scatters the light waves, yielding patterns of light intensity along two axes — one parallel to the laser source, and one at right angles to the source which detects light scattering. These patterns, in turn, can be analyzed to ascertain properties of cells, proteins, and other molecular entities in the sample source. The actual images generated in this process are not literal photograph or microscope-enhanced pictures, but rather are composite representations of light intensity formed by plotting parallel and scattered light patterns on a two-dimensional axis (effectively, a high-density scatter plot). While these representations are technically diagrams rather than images, certain image-analysis concepts have analogs in this context; for instance, "gating" — selecting image regions of interest (e.g. as measures of cell count for different varieties of cells) — is analogous to image segmentation.

Flow cytometry data is usually presented via **FCS** (Flow Cytometry Standard) files. The most comprehensive package for working with the **FCS** format is based on **R**, although one **C++** library (developed by the Fred Hutchinson Cancer Research Center) has been isolated from the surrounding **R** package into a library that can be compiled standalone. Outside the **R** context, however, this library has limited functionality for applying predefined gating schemes or image visualization; one goal of **CR2**, then, is to enhance the relevant **C++** package with added tools allowing the code to be used in application contexts other than **R**.

Meanwhile, other biomolecular visualization technology — including "image cytometry," an older but still useful alternative to flow cytometry — require more conventional image-analysis techniques, so that **CAPTK**-style image-analysis pipelines are appropriate in these contexts. Different imaging modalities, including bioluminescence, scans, and s, have been used for immunological

cellular-scale imaging.[14] A multi-disciplinary toolkit for immunology-related image processing should include the framework for incorporating Machine Vision algorithms specific to image processing in the context of immune-cell visualization as well as indirect cellular analysis enabled by methods such as cell cytometry. The CR2 code will try to move in that direction by pairing cytometric libraries with an overarching image-analysis context.

Vaccine Data and Geographic Information Systems

Another area where vaccination data intersects with image-processing is that of using Geographic Information Systems (GIS) to document vaccination campaigns. Geospatial mapping has been used to support vaccination logistics in several ways. The most straightforward is to map where vaccines have been administered, though some projects described by groups such as GAVI (the Global Alliance for Vaccines and Immunization) have used mapping technology to represent the location of clinics, and to visualize demographic or jurisdictional data that factors in to advance planning for vaccination campaigns.

Geospatial mapping typically starts with an underlying image — specifically a geographic map, which may be constructed by software, derived from satellite or street-view photography, or some combination — and then superimpose on the map-image a data structure identifying geospatial points or regions of interest. Numerous software packages exist to construct such annotated maps, such as (in the QT context) ARCGIS and QGIS. In general, using these libraries involves writing plugins to import domain-specific data which includes geotagging, along with relevant details for each location. These plugins therefore provide data structures that supply GIS coordinates for sites of interest, along with structured data about the significance of each site; when the map is visualized, this supplemental data is converted to markings or text viewed as superimposed on the map image, providing helpful information to those studying the map (either for navigational purposes or as a visualization of geospatial data).

In the vaccination context, geotagged annotations might involve color overlays documenting how extensively a vaccination campaigned covered each locale within its target area; or they might identify fixed locations, such as schools or clinics where a vaccine was administered. Several such maps have been developed for campaigns in particular countries, or other geographic areas, but there does not appear to be a general-purpose C++ library for modeling and then geotagging vaccination data so that the resulting data collection can be used to populate a GIS plugin. Therefore, the vaccination/immunology toolkit described in the previous subsection, providing code for integrating immunological and image-processing data, will also include the framework for vaccination-related GIS plugins.

Part V: Conclusion

Most of the code described here will be developed in conjunction with the Covid-19 Elsevier Volume, in part coinciding with the publication of the book and continuing to be refined and expanded after the fact. Some of the more technical programming areas, such as cytometry and image processing libraries, will hopefully be generalized to multi-purpose toolkits not limited to the Covid-19 context. Feedback on CR2 overall, and the more specific technical areas in particular, is welcome.

For more information please contact:
Amy Neustein, Ph.D., Founder and CEO
Linguistic Technology Systems
amy.neustein@verizon.net • (917) 817-2184

[14] See https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6689497/ or https://mbio.asm.org/content/10/5/e00317-19.