A Proposed Software Development Kit for Proteogenomics, Tumor Microenvironment, and Cellular Systems Research

Fully leveraging the APOLLO networks
and similar molecular-biology data-sharing initiatives

I Overview

Linguistic Technology Systems (LTS) proposes developing and curating a suite of software tools integrating several digital resources supporting molecular biology and proteogenomics, such as those comprising the APOLLO (Applied Proteogenomics OrganizationaL Learning and Outcomes) network. In particular we propose a Standard Development Kit (SDK) that would integrate functionality exposed via the GDC (Genomic Data Commons) API, Data Model, and UI toolkit, along with the Proteomics Data Portal UI toolkit and the Cancer Imaging Archive (TCIA) API. Our goal is to identifying overlapping functionality within these three data networks — which collectively supply the APOLLO data ecosystem — so as to implement software tools which would access any or all of these networks in the context of research combining proteomics, genomics, bioimaging, and clinical data. Such tools could add functionality to the existing software ecosystem supporting APOLLO that could streamline research workflows at several stages, including data acquisition, analysis, and publication.

II Addressing Limitations in the APOLLO Software Ecosystem

2.1 The Benefits of Embedding APOLLO Support in Science Applications

The three APOLLO networks — GDC, TCIA, and CPTAC (the Proteomics Data Portal) — all provide some tools to help researchers submit and acquire data sets. Most of these tools, however, are exposed as web services rather than as code libraries that could be bundled into scientific applications. For example, the GDC provides a property-graph based data model which integrates molecular, clinical, and genomic data according to predefined data types and interconnections. This model is instantiated within tools used by the GDC to validate and harmonize submitted data sets prior to their being made publicly available (data sets may be submitted to GDC either through the GDC API or via an online portal). However, the GDC does not provide software tools or code libraries to facilitate the implementation of computer applications which would curate data submitted to and/or acquired from GDC, so that researchers could perform their own validation steps in preparation for the

GDC analysis. The official GDC API client is a guide for programmers who wish to generate requests against GDC endpoints, but cannot be used directly to access GDC data unless applications embed a Python interpreter. Similarly, the GDC User Interface components — which are intended for software providing visualizers for GDC data — are based on JAVASCRIPT. In short, the GDC data model, API client, and UI toolkit are each targeted at different programming environments, and GDC themselves do not provide a unified framework which integrates these areas of functionality into a common programming environment. The GDC's published code and web services document the requirements for applications seeking to interface with GDC data submission, validation, and acquisition protocols, but it is left to third-party software to unify these capabilities into a single platform.

With respect to CPTAC and TCIA, both of these networks require a multi-step data-acquisition process which could be streamlined with the help of software components. The TCIA API is split between two interfaces, one of which may be accessed via a client library provided in JAVA and PYTHON, but as with the GDC API these tools have limited value for standalone biomedical applications. Considering that proteogenomics research often will access both cancer imaging and genomic data, it would be reasonable to provide a full-coverage API client for both GDC and TCIA, using similar programming methods to access each of the API endpoints provided by either organization. Such a client could be implemented as a C++ code library able to be included directly as source code into scientific/biomedical applications and/or project-specific code, and could be wrapped for languages such as PYTHON and R as needed. Similarly, the UI tools provided with CPTAC could be generalized to an integrated proteogenomic toolkit combining the CPTAC and GDC UI components. The combined code base would then be available as a suite of GUI classes that could be embedded in scientific/biomedical applications.

In the case of CPTAC, the UI code is used to provide data visualization capabilities in conjunction with downloaded CPTAC data sets. For example, downloads acquired through the Proteomic Data Portal will include HTML files providing interactive plots and other figures summarizing information encoded in the rest of the data set. If CPTAC support is encapsulated in a module embedded in scientific/biomedical applications, similar visualization files can be targeted at the host application, allowing users to visualize the CPTAC data summaries directly rather than opening a separate web browser to examine the HTML files generated for a downloaded data set.

2.2 Using an APOLLO SDK to Document Published Research

While an integrated data model, API library, and UI toolkit can make access to the APOLLO networks more convenient for researchers and programmers (which can benefit proteogenomic and related research), a unified proteogenomic SDK would have the added benefit of documenting research methods and workflows more thoroughly than is possible with the existing APOLLO framework. The current APOLLO code libraries and services are scattered across multiple components within three distinct networks, and much of their functionality is exposed via web portals intended to be manually visited by individual researchers. These factors make it difficult to model and record the specific data submission and acquisition workflows which underlie research that leverages APOLLO. Interfacing with APOLLO via computer code aligns more with research transparency, replication, and data-sharing goals/protocols such as Research Objects or the FAIRsharing initiative. In lieu of individual web sessions, accessing **APOLLO** via **API** client libraries would allow each research projects' code and procedures to acquire and/or submit **APOLLO** data to be publicly examined, shared, and reused. The code which individual research projects use to interface with **APOLLO** could be published in data sets or supplemental material associated with research papers, which would present a more structured and reusable representation of research logistics than data-acquisition summaries included informally in publication text.

In short, a proteogenomics-oriented **SDK**, whose functionality could be based on and informed by the **APOLLO** components, would help consolidate research methods and data management for proteomics, genomics, and related disciplines by modeling common research practices, workflows, and software design patterns. This could help synchronize, accelerate, and synthesize research in these areas much as platforms such as BioConductor or GaTK have served to centralize research in fields such as oncology and gene sequencing. Contemporary scientific guidelines encourage authors to combine raw data, supplemental text, and computer code into self-contained bundles — for example, Research Objects — which can be shared, cited, and evaluated as a single research unit, analogous to an individual publication. Research Objects should be both *integrated* (combining all code, data, and text/documentation into a single resource) and *self-contained* (providing all necessary components for researchers to access and examine research objects within the research object bundle itself). Achieving these goals is more difficult in the context of intrinsically multidisciplinary areas, such as molecular biology (especially tumor microenvironment, proteogenomics, cellular systems simulation, etc.) where data may be obtained from multiple sources (with differing accession steps and preparatory code) and/or analyzed or visualized within multiple software applications. Integrating multiple data sources and supporting code libraries into a single **SDK** can help scientists document multidisciplinary research through integrated and self-contained bundles (such as Research Objects) which conform to contemporary data-publishing standards.

The rationale for encouraging publication of integrated and self-contained Research Objects lies not only with added convenience for other scientists who may seek to evaluate, reuse, and/or replicate research methods and findings; it also serves to clarify and delineate the specific methods and claims advanced by each particular research project. A self-contained code/data archive, unambiguously tied to a single research project, can serve as a canonical documentation of the research protocols adopted for that project (including data acquisition/preparation and analysis, implementation of supporting code/assumptions, data models and statistical assumptions, etc.). A Research Object bundle, or similar research-documentation archive, can therefore augment the value of scientific publications by presenting to the scientific community a specific resource associated with but independent from particular academic papers, a resource which systematically and operationally illustrates the theories, claims, and methods described informally in the accompanying publications. In short, Research Objects are intended not only to share research methods for the sake of transparency and replicability, but to produce a singular resource encapsulating the full logistics and scope of a research project's code and data. Consequently, it is important for software tools facilitating scientific research to promote illustrative and self-contained Research Objects fulfilling these criteria. In the proteogenomics context, for example, research work would be more readily consolidated into self-contained Research Objects if research were conducted through an integrated **APOLLO**-oriented **SDK** rather than through the existing **APOLLO** tools, which are disconnected from one another and targeted at separate networks.

2.3 Implementing a Unified Graph Query Language

We also propose a novel technical infrastructure to drive this proposed **APOLLO SDK** and related tools, addressing concerns such as data modeling/integration and query evaluation. In particular, Graph Query Languages, such as **SPARQL**, **GML**, **GraphQL**, and Gremlin, are increasingly being used to define and filter data acquisition parameters and data integration criteria; for instance, **GDC** uses **GraphQL** at the core of its query interface, and the **BioNetGen** cellular simulation system (discussed in the next section) encodes data structures via **GML** (Graph Modeling Language). However, there are several alternative languages for querying and describing graphs,[1] which can hinder efforts to integrate multiple data sources into a single **SDK**. We propose to address this limitation by implementing a unified Graph Query Language that recognize distinct graph metamodels (hypergraphs, property graphs, Semantic Web graphs, etc.) and which is optimized for embedding in software applications and code libraries. The premise behind this proposal is that recurring patterns in research workflows and data acquisition/management requirements can often be modeled in terms of hypergraph data structures, so that tools for manipulating hypergraphs can provide a foundation for implementing recurring concerns in research-oriented software. In particular, we propose a Property-Hypergraph Virtual Machine (**P*h*VM**) that would serve both to evaluate property graph/hypergraph queries and to constructs objects or procedures facilitating research-workflow concerns such as distributed/remote procedure calls or exporting data to a data commons.

For example, Distributed Computing or "remote-procedure" protocols come into effect when one software application or code library need to invoke a computational process or capability which is provided by a different application, or by a service hosted on a different computer, or in general is executed within a computational environment distinct from the application's current state.[2] In order to systematically identify the sequential steps or operations typically required for this setup and processing, our Property-Hypergraph Virtual Machine under development would identify units of functionality applicable to different stages of distributed/remote procedure processing as one group of operands exposed to a Graph Query Language, insofar as data structures that can be constructed to manage/initiate distributed/remote procedure calls can be modeled as hypergraph structures.

In addition to distributed/remote procedure calls, similar operand-groups within the Property-Hypergraph Vir-

[1] Partly because different graph database engines have distinct structural designs — the generic term "graph" encompasses many different computational models.

[2] Distributed/remote procedures have some analogies to ordinary procedures — for instance, they can have input and output parameters — but they require a more complex setup. For example, the result of a distributed/remote procedure may be provided in the form of a file that has to be parsed and processed, rather than a single value that can be used directly. Also, there may be a time delay while the procedure is remotely executed, so the calling code often uses a signaling or callback mechanism — something more complex than just waiting for a procedure result — wherein a a procedure to process the remote procedure's results is separate from the procedure which invokes the remote procedure to begin with. As such, constructing a distributed/remote procedure call requires coordinating several different areas of functionality — encoding parameters, delegating handlers to process returned data, parsing results, extracting values from the parsed data, and so forth. Well-designed code libraries will organize functionality and exposed procedures around recurring patterns, such as the setup and processing of distributed/remote procedure calls.

tual Machine model Domain Specific Language parsers, data-export functionality, and other research-workflow requirements in terms of hypergraph structures, providing a framework which formally identifies recurring patterns in research-oriented software concerns. We believe that a query language based on this Virtual Machine could therefore add value to an APOLLO SDK insofar as it would not only provide graph query capabilities but would also identify and consolidate research-workflow requirements, to facilitate the implementation of integrated and self-contained research software.

III Combining an APOLLO SDK with Cellular Network Modeling

Aside from the research benefits identified in the prior section, an APOLLO-oriented SDK could establish coding practices, particularly in the areas of molecular/systems biology and precision medicine, which would apply to other software projects apart from those explicitly targeting APOLLO. For example, the National Center for Multiscale Modeling of Biological Systems (MMBioS) provides a platform for research involving cellular systems, molecular modeling, biomolecular simulations, and image processing. Their stated computational goals to provide new features and capabilities include developing a new database and API for cellular networking modeling (supporting their widely-used BioNetGen software) and expanding APIs for cellular-scale geometric and physical simulations. Both MMBioS and APOLLO illustrate cross-cutting concerns with respect to data management, curation of data-set archives, API design, data models and validation, and UI development. These concerns and requirements in the context of scientific/biomedical data-sharing initiatives are interconnected in ways that can reappear across different projects, suggesting that certain overarching design patterns may be leveraged by software tools that can be useful when instantiated for distinct specific projects, such as APOLLO and MMBioS.

3.1 Proposing an Network Modeling SDK for MMBioS

In the context of MMBioS, a C++ API (identified as a project-aim in the MMBioS Network Modeling research area) would permit BioNetGen actions to be called directly rather than through a command line, though perhaps the API could be designed (as is a popular pattern in scientific applications) to support workflows that can be manifest either through command-line actions or directly in code sequences (as well as indirectly via remote procedure calls). In addition to generating command-line invocations, the **BNGAction** Perl Module (used to access BioNetGen functions for data management and analytics) does rather detailed preparatory checks in some cases, so equivalent functionality would have to be implemented as an API layer. In other words, a C++ API would presumably have several stages, with preliminary logic at one stage yielding data structures to a second API layer that communicates directly with underlying C++ code.[3] There are numerous options for mod-

[3] That is, C++ data structures would be generated in lieu of command-line invocations, though at this step it would make sense to allow such data structures to be serialized into workflow descriptions, executed indirectly via command line or RPC and other deferred/distributed methods, etc., as well as being executed directly.

eling distributed/asynchronous procedure requests (e.g. whether the response requires a separate parsing/value-extraction step, whether response callbacks carry state, and the specific reactive/function-object mechanisms are used to supply callback procedures); a rigorous **API** should allow each of these options to be employed when appropriate and should clearly define the protocols and requirements in each case. It also appears that much of the data visualization associated with **BioNetGen** runs through a similar Perl/command-line pipeline as BNG Actions, so presumably the **API** could support visualization, perhaps expanding the range of visualization outputs (maybe full-fledged **C++ GUI** components instead of text formats such as **GML**).

3.2 Modeling Research Workflows via Database Engineering Technology

With respect to the overall goal of curating a cellular-systems-oriented database, components driving the **BioNetGen API** could be reused in the database implementation directly, insofar as the basic units of persistence within such a database would typically be complex objects within significant internal structure. This means that conventional layout and query models, such as **SQL** and relational tables — or even **RDF**-style labeled graphs — would not work conceptually against this sort of data. Instead, the engine would need some layer of complex objects to serve as intermediaries between applications and the database, both for receiving/updating new data and retrieving data via queries; such intermediary **C++** classes could be unified with **API**-processing layers in a common code base. Similarly, the code base could include parsers for a special query language that could be integrated with **BNGL** and with workflow description languages such as **CWL** (Common Workflow Language). Finally, we propose to engineer the relevant database components with an emphasis on supporting multiple stages of the research process, including publication-related actions such as generating datasets and using in-document microcitations to connect datasets with article text.[4]

In sum, there are six or seven facets of data management which come to the fore with database systems in the context of application integration, **API**s, scientific data curation, and publication/dataset management – parsers for special-purpose languages; domain-specific query evaluators; custom **GUI** components; interacting with analytic capabilities both in-process and out-of-process; metatype systems allowing software components to model scientific processes/phenomena; application integration via type-level serialization and persistence; and dataset/publication integration. This set of concerns reappears in numerous scientific-computing contexts (one can identify similar patterns in bioimage processing, for instance) which suggests that they may serve as a general architectural framework for scientific-computing database and application implementation.[5] In the context

[4] Ideally, publications, data sets, and analytic procedures/simulations could be unified into Jupyter-style interactive presentations. Data visualization could thereby be integrated into **BioNetGen** research using software tools similar to our proposed integration of Proteomics Data Portal **UI** tools into an **APOLLO SDK**. In particular, the **APOLLO** and **BioNetGen API**s could share a format for representing data visualization assets (such as plots and figures) via individual files that could be included in shared data sets, to be interpreted as **GUI** resources by applications which embed the **API** client libraries.

[5] Cellular systems represent one area where database requirements encompass heterogeneous data types with often complex interrelationships, but there are similar examples in the overall context of molecular or systems biology, such tumor microenvironment simulation and proteogenomics (which unifies protein and gene analysis as well as bioimaging and clinical data, and therefore intrinsically represents an integration of heterogeneous data profiles).

of publishing research work and data sets, such a framework could help scientists conform to guidelines such as FAIRsharing or the Bill and Melinda Gates Foundation Guidelines for Authors. In the context of scientific computing, such a framework could help implement code which documents scientific methods and assumptions — for instance, explicitly identifying scales and units of measurement and admissible ranges on data fields, or procedural pre- and post-conditions.

With respect to hypergraph data modeling, hypergraphs have already been used in numerous algorithms and research projects to model complex interactions in molecular biology, cellular systems, and related fields. Hypergraphs are a natural generalization of both Semantic Web data and property graphs. Numerous facets of biomedical data management are naturally modeled via hypergraphs.[6] These represent different kinds of data structures which embody recurring design patterns and can form the basis of a distributed object system (which could be called a Distributed Hypergraph Object System, or DHOS, to emphasize the use of hypergraphs to model/serialize these structures). DHOS objects are not the same as C++ objects, but DHOS objects serve as a nexus connecting multiple C++ objects. As such, support for the intersecting data-management concerns mentioned above can be provided via different kinds of DHOS objects and tools for connecting ordinary C++ objects to the relevant DHOS objects.[7] Such C++-to-DHOS connections can then be modeled as part of a C++ runtime reflection system and provide a basis for defining the sort of runtime data that should be exposed through such a system. Our Property-Hypergraph Virtual Machine (P*h*VM), mentioned above in the context of graph query evaluation,[8] is connected to DHOS insofar as one cluster of VM instructions involve steps to construct DHOS objects and assert connections between DHOS and ordinary C++ objects.[9]

In short, we have identified what we believe are several recurring patterns in database engineering and cloud-computing requirements for biomedical research and clinical data (encapsulated in the idea of a "Distributed Hypergraph Object System" minimally sketched above) and we believe it would be beneficial to establish a software development environment which models and operationalizes these patterns directly. Moreover the catalyst for engineering such an environment should be specific projects such as this proposed contribution toward the network modeling database and API, using real-world information and feedback to refine the environment's tools and capabilities. This is why we are hoping to contribute code toward the database and API implementations and/or supporting ecosystem as explained in this email and our prior communications.

[6] For example, parse graphs for formal languages, semantic infosets which represent the value payloads instantiated via parse graphs, query-aware serialization, and reactive/asynchronous procedure requests (or meta-procedures, in workflow-oriented terminology), including distributed/remote procedure calls as discussed earlier.

[7] Parse graphs, infosets, stateful or stateless asynchronous procedure requests, and query-aware serializations, as these various structures are called for the purpose of specifying DHOS, are examples of DHOS kinds.

[8] For example, P*h*VM is designed to implement most steps of the Gremlin VM, providing Gremlin-style graph query and traversal among other graph manipulation paradigms.

[9] In general, P*h*VM is divided into six areas: graph query/traversal; DHOS; scope/stack manipulation; built-in operands; function calls to kernel C++ procedures, user-supplied C++ functions, and/or procedures defined in P*h*VM itself; and function calls through the Qt metatype system (along with related Qt integration features, e.g. signal/slot connections and Qt properties).