APDL Introduction

APDL ("Application Provenance and Description Language") is a tool for describing computer software applications and the data which they generate. APDL is designed for environments where multiple applications exchange services and contribute data to a shared pool or "lake". In these scenarios, information about *data provenance* — which application had produced a given data aggregate and when — may be essential for subsequent viewing or processing of that data.

Moreover, in "linked data" networks, connections may be asserted between data generated and governed by different applications. Because of such "cross-application" connections, users may discover data while using one application which must be accessed through a different application. For a concrete example: audiology and neuroimaging tests may be linked by indicating a common diagnosis, but reviewing the test results demands different software for the audiological and neurological data, respectively. Modern data stores are designed to combine heterogeneous data sources, such as test results from different medical fields. But we must not only *store* and *analyze* data; we must also ensure that data is accessed and visualized through correct sofware.

In linked data, an application may be working with a node containing data which it *does* understand but then discover other data connected to this node whose format or file type it *does not* understand. Continuing the audiology example, reports from multiple sources — such as occupational/physical therapy progress notes or neuro-imaging scans — may be linked to hearing tests, performed on the same patient around the same time. Despite their mutual relevance, however, the neuro-imaging software (for example) may lack algorithms to parse the audiological files. This supplemental information is therefore "opaque" to the imaging software. Instead of just ignoring this opaque data, APDL allows applications to *find* (applications), *share* (data and contextual information), and *delegate* (computations) with other applications.

APDL therefore includes a framework for sharing "requests" as well as data, via an inter-application communication portal called a "Public Service Interface" (PSI). Through PSIs, applications may launch other applications and request specific actions — such as opening a file of a type understood by the requested application but not the requester. At the same time, APDL allows each application to be summarized, so that both developers and end-users understand the data and service exchanges that occur through the resulting multi-application networks.

APDL models several kinds of information, which will be outlined here. That outline will be preceded by a summary of why APDL can be useful for institutions, and then followed by a review of APDL's deployment, standardization, and addition developer tools that support APDL.

I. Use Cases for APDL

In its totality, APDL provides both summarial, high-level overviews of applications and finer-grained descriptions of GUI elements, datatypes, and functionality. Therefore, APDL may be useful both to software developers looking for rigorous code documentation (and in some contexts code generation) tools, and to administrators keeping track of software as part of an institution's assets and research tools. Possible scenarios where APDL may be used include the following:

1 Administration of Digital Assets

Many organizations need to maintain an inventory of digital assets and collections — which can include files, databases, and web services as well as individual applications. Digital Asset Management tools can be useful for cataloguing electronic resources, but do not model applications themselves in detail. As a result, APDL can enhance Digital Asset Management both by

attaching extra meta-data to applications seen as digital assets, and by associating other assets (such as individual files) with applications related to them (for example, identify which application can open which files based on file-type). APDL enables "Application Description as Asset Management" wherein organizations identify software/version/platform combinations used by their community, to help standardize digital assets for the best possible interoperability.

2 Multi-Application Documentation

Quality documentation is often an underprioritized step in the software engineering process. Organizations will often maintain informal guides (for example a professor may publish instructions on how to use software needed for a science course), but institutional users of software should aim for a more systematic and comprehensive approach. Because applications are digital assets, tools which make software more user-friendly enhance the value of these assets.

For instructions and documentation, by far the most common coding practice is to direct users to external web or PDF browsers — often the web site where users download the application in the first place — but this eliminates any channel of communication between the software where users read about technical applications and the applications themselves. A more user-friendly manual would instead be networked with the application, passing information and requests to a running instance of the application. Users then have an option to preview a feature in the running application which they are reading about in the manual.

As a concrete example, if users are reading about how to choose a color via an HSV (hue-saturation-value) color wheel (which is more complex than other color-select dialogs), the manual viewer software (or "*eReader*") can request the running application to actually open an HSV dialog. This example also provides a case-study of applications serving *requests* posted by partner applications — that is, a Public Service Interface (PSI). In this case, the PSI requestee would recognize data sent by other applications which encods a request saying, in effect, "show an HSV dialog" (a variation would be to request that the requestee send back to the requester a code representing a color the user then selected from the dialog, if any).

Since documentation enhances the value of digital assets, documentation tools and artifacts (like instruction manuals in PDF format and structured data used to enable cross-application communication serving the goal of elucidating application usage and capabilities) are digital assets in their own right. Ideally, Documentation Artifacts can be stored along with other kinds information in portals like a Semantic Data Lake (SDL) or Linked Data Set.

3 Publication and Document Management

The computer software used to gather and analyze data, create visuals, or run experiments presented in a publication, is an important piece of information pertaining to a book or article. Publishers and authors have several incentives to curate this information:

3.A Document Search: Readers may want to use application-related data as a criterion for finding publications whose methods or subject-areas fit their interests. For online libraries annotated via APDL, a document search portal could include an option to search for papers (and where applicable their embedded datasets) on the basis of applications used to generate/curate data.

3.B Reproducibility: Publishers are increasingly encouraging authors to publish data sets and otherwise prioritize reproducibility. This can include providing information in a structured, searchable format about the applications used to process data and graphics. Adopting such a practice makes the author's methods more transparent, and helps accelerate the preparations that others have to put in place when trying to replicate the author's results.

3.C Data Sharing: For documents which reference public data sets — including digital publications which embed data sets directly as electronic assets — APDL can help readers access these embedded or linked assets. Software for eBook and PDF files can use APDL to deal correctly with embedded files of different types. That is, document viewers or eReaders which belong to multi-application networks can help users access embedded data, by passing it to partner applications that can open embedded files in scientific or technical formats (by contrast, conventional PDF/eBook readers either ignore embeded files entirely or require users to manually extract and save them, followed by manually launching a second application with which to open them).

3.D Application Enhancements: The partnering of PDF/eBook readers with scientific/technical applications can also benefit users and developers of technical applications, because publications — including manuals and tutorials as well as research papers — can help users understand how to operate the software. Applications can link to documnts which explain the terminology and analytic capabilities that they provide as well as step-by-step usage guides.

The partnering of technical and eReader applications in a multi-application network can improve documentation, which benefits end-users, as was discussed in the previous item. In synopsis, publications — especially academic, scientific, and technical documents — are most valuable to users when document viewing or "eReader" software is fine-tuned in coordination with scientific and technical applications and with organizations that administer many software components on an institutional level, linked via data portals and protocols.

Unfortunatly, application information, application-level meta-data or data provenance, and application documentation (for developers and end-users) — that is, the key space of concepts modeled by APDL — are not represented in sufficient detail in traditional scientific and biomedical data sharing initiatives: that is, in the level of detail needed by application developers trying to interoperate productively with other applications. APDL redresses these lacunae by being grounded on application-development experience and technologies directly (like Qt) rather than (for example) database management or analytics — disciplines whose techniques come into focus only after applications have already been implemented and provisioned.

4 Resource Bundling

For sharing information, APDL adopts a concept of "Portable Resource Bundling" to work around situations where individual applications often have more exacting protocol and representation demands than do affiliated data networks themselves. Portable Resource Bundling preselects information and assets from a heterogeneous data store, making this selection available to applications which are not required or able to query the data store directly. As a motivating example, imagine embedding files and data into a PDF document, which provides a convenient vehicle for sending the resource set to an end-user. The PDF viewer then needs to be extended with libraries to extract and process the embedded data, but preselecting resources eliminates the need for the PDF viewer to have network access and programming capability to access the data store directly. The PDF viewer instead extracts data only from the Portable Resource Set, which (unlike a persistent data store) can be structurally optimized for application integration.

Portable Resource Sets allow applications to share data via heterogeneous platforms, like a Semantic Data Lake, while still focusing on the technical field and business uses for which the application was designed. It is unrealistic — continuing an earlier example — to expect software in an audiologist's or physiotherapist's office to directly access affiliated hospitals' Semantic Data through a bonafide semantic (SPARQL/Ontology/Semantic Web) interface. Nevertheless, isolating software in the confines of a single office is not an option either, supposing the goal (in the context of health care) of integrated patient care. Standardized vocabularies, like EHRs,

allows data sharing between and among "hubs" (like hospitals) and "spokes" (like outpatient services), but this gives peripheral offices access only to generic kinds of information through web-oriented portals, like REST APIs. From the periphery, sophisticated information systems like SDLs end up being de facto web applications, losing most of the benefits of Semantic technology.

APDL helps redress this situation by organizing resources so as to simplify the selection of "portable" resources intended for a specific application. Via data provenance, APDL allows resources to be identified which are direct serializations of data types produced by the target application or libraries which it uses; it also allows resources to be identified which encapsulate files whose type is recognized by the target application. These are called *priority resources*, and data portals can embed special processing units that, via the APDL ontology, identify priority resources for a particular application and can select them from a data repository and unify them into a portable set, like a zipped folder. APDL can then help the target application unpack the set into files and in-memory data which it is capable of working with directly.

5 Graph Compilers

Most compiler technologies have some stages, or components, which work with graph structures, but these are usually tangential to the main compilation process (which tends to work on different sorts of representations, like Abstract Syntax Trees). That is, certain structural representations of computer code at various stages of transformation are typically studied as graphs, but this tends to be for specific compiler tasks that get folded in to the overall compiler stack.

By contrast, there is an emerging field of "graph compiler" technology which features graph representation as a more central and persistent aspect of the compilation process at all stages. These perspectives treat compilers as working fundamentally on graphs at every step, and expresses the tasks and questions which compilers must resolve as fundamentally questions about graph structure and graph transformations. This methodology is guided by the idea that graph representations provide a more elegant or formally useful view on computer code than such alternatives as trees or sequences — especially given that "graph representation" encompasses a range of systems (hypergraphs, labeled ordered graphs with various added structures, acyclic graphs, multigraphs, bi- and tripartite graphs, and so on).

The emergence of Graph Compiler technology comes from multiple sources — including the Semantic Web, which suggests the possibility of approaching Software Language Engineering through broad Semantic Web data types and query mechanism, integrating programming languages and compiler technology with the overall Web ecosystem; and Artificial Intelligence, where "Graph Compilers" power a new generation of neural networks and concurrent processing cores. The technologies associated with these diverse trends are not identical, but they do suggest the gradual consolidation of "Graph Compilers" as a distinct theory and practice.

Some of the tools associated with APDL — such as DynamoIR, or "Dynamic-Native-Modular Intermediate Representation" — can be described as contributing to a "Graph Compiler Framework", albeit one differing in goal from the Semantic Web and AI trends. The key priority of graph representation in this context is to analyze all of the various "channels" via which a computation (a function or procedure implemented in a programmin language) can communicate and influence its external environment — its computing environment and even, in the case of CyberPhysical systems, its physical environment. A full compiler technology is beyond the scope of APDL — compilation requires modeling and transforming all parts of computer code, not just selected features relevant for documentation or a Public Service Interface. Nevertheless, the technologies used to generated APDL for the more selective descriptive and information-sharing contexts

applicable to APDL can be extended to a more fine-grained code representation. In this sense APDL-related tools may be of use to those intersted in Graph Compiler solutions.

6 Application Development

Programmers can embed APDL in any Qt application or any application which can link against the Qt libraries (this includes the option of licensing a suite of development tools from Linguistic Technology Systems, Inc. which includes APDL support, but no commercial license is required to use APDL as a description and annotation language). While APDL can help with certain programming requirements within a single application, the more significant benefit is that software which provides and understands APDL data is poised to join with other applications into networks that collectively implement workflows. Via APDL, applications *discover* and then interoperate with other applications, thereby realizing *find, share, and delegate* workflows. This ensures that each single application's capabilities can be enhanced without additional coding or loss of focus from an application's primary purpose (that is, without "feature creep"). Applications can request other applications to provide capabilities they themselves lack, while still being able to influence partner applications' behavior. Integrating and networking with partner applications is intermediary and a compromise between simply launching external software as a separate process and reusing external software components as embedded libraries: developers gain some use of third-party libraries without added build and maintenance complications.

II. The Components of APDL

The data generated with APDL has two different levels: information about applications themselves and information about individual data aggregates. In APDL, the former is referred to as "description" data while the latter is "provenance" data. While they encompass different concepts, description and provenance data are interconnected — by linking provenance to application info, heterogeneous data stores acquire new workflow capabilities. Formally, these links are established via application identifiers embedded in provenance data.

With this in mind, the four top-level components of APDL can be broken down as follows:

1 Provenance for Individual Data Aggregates

This is APDL data that can be attached in principle to any arbitrary data structure. In practice, the provenance information will have a different form depending on whether the underlying data is *application-grounded*; that is, whether it comprises a single instance of a data type recognized or implemented by the application. Applications are free to use APDL with non-grounded data, but other applications using this data (the "partner applications") will have fewer processing options for non-grounded as compared to grounded data. The APDL provenance information can be joined with underlying data using any provenance-representation mechanism native to the relevant data store, such as RDF triples or annotations on triples.

2 General-Purpose Application Description

This is information which uniquely identifies an application so that applications and data can be unambiguously joined in provenance annotations. The goal of the *overall* application description — in contrast to lower-level details outlined in the next two items — is to give developers the information they need to work with data generated by other applications with which they

share a data pool. This includes information on versioning, supported file-types, and tips for developers: instructions or comments about building, launching, and/or communicating with a running instance of applications, including project-specific observations in the specific context (e.g., operating system and hardware) applicable for their organization.

3 Application Description at the User Interface Level

This information allows applications to publish a "catalog of features" that may help end-users to learn about new applications. In a multi-application network, users may switch from an application they know well to a new application based on the idea of shared data discovery; this puts an onus on developers to anticipate users' questions and concerns. In particular, users can often describe the action they want to perform without knowing how to invoke that action through a GUI. For these scenarios, APDL can be used to list both user-visible operations (any action the application can take, on explicit request from the user) and user-visible GUI elements (windows, buttons, menus and menu items, etc.). These lists can be helpful because end-users may not realize how GUI elements are mapped to operations. By rigorously documenting such mapping, APDL can serve developers who want to employ a standardized documentation for multiple applications. For example, many operations — such as saving a file in a different format — are supported by many applications, and a multi-application environment can provide a unified search feature which recognizes functionality shared by many applications as well as operations endemic to one single application.

4 Application Description at the "Public Service Interface" Level

This includes documentation of functions implemented by applications that can be requested externally (and potentially other functions that are not PSI-visible but which help support PSI requests). This branch of APDL includes both human-readable information (for developers) and structured data compiled for code generators.

Some of the concepts modeled via PSI descriptions derive from formal systems like the π-calculus, or a partially new variation via a "channel algebra". Channel Algebra (in the present context) presents functions' behavior and specifications with semantics that serve as a fine-grained type system (with features like "dependent" types and function-calls through a network interface), while also modeling the differences between the family of function-types thus narrowly defined and the coarser types expressed by most concrete programming-language type systems. This more mathematical subject matter is outside the scope of the present paper, but can be discussed in greater depth with those interested in using APDL for workflow systems.

5 Graph Serialization Protocol Description (GSPD)

The concepts modeled by GSPD pertain to whichever specific persistance technology is targeted by the current application using APDL. The goal of GSPD is to negotiate the diversity of graph/RDF storage systems, providing both an abstract Graph Serialization Protocol and a description language identifying technical requirements that may vary between databases/data stores. Developers can therefore focus first on building graph structures in a database-neutral format, while separately using tools that commit the general-purpose graphs to persistant storage — recognizing variations in database storage model (in-memory, file, shared-memory, distributed); annotation format (URL-scoped, character string, or in-memory objects); supported definition and query languages (Turtle, N3, SPARQL, Cypher, GraphQL, XML); storage units (triples, quads, linked records, hypergraphs); and communication protocol (in-code, queries, HTTP).

III.

Standardization and the Reference Implementation

Data sharing protocols are frequently based on a particular data serialization language, like XML, JSON, or RDF — often paired with specifications, like OWL Ontologies with RDF, that determine when documents that intend to encode data according to the protocol are well-formed. A risk in this approach is that designers become preoccupied with details of serialization and markup format, distracting from the structure of the data being defined. As a result, APDL does not specify any particular data exchange format — or, to be precise, APDL metadata in itself is orthogonal to format, although APDL can identify the formats used by modeled data (that which is *described by* APDL, in contrast to APDL metadata *doing the describing*). Instead, APDL is modeled as a data type using the Qt/C++ libraries. Any APDL structure is (or can be converted to) an instance of this type. Developers can obtain a code library used to incrementally build these data structures and share them in a binary format. Developers are free to serialize APDL using any variant of XML, RDF, S-Expressions, or other markup or expression languages.

Accordingly, APDL data is considered to be correctly shared if the sender and receiver applications end up with functionally identical copies of the serialized type-instance, irrespective of the exchange format. This will unambiguously hold if both applications use the same code library (which includes the option of the Reference Implementation being exposed to other languages in conjunction with Qt bindings). A non-Qt or non-C++ implementation can be considered "functionally equivalent" to the Reference Implementation type if code is explicitly provided mapping to Qt data or if it seems clear that such conversion code would be straightforward to implement.

Basing APDL on Qt is consistent with APDL's primary use case as a data-sharing protocol for desktop-style applications. Using Qt introduces no onerous external dependencies, and makes it straightforward to deploy basic cross-platform, GUI-based executables providing information about APDL data. Also, Qt has mature and robust support for TCP/HTTP networking, launching system processes, manipulating binary streams, and other tasks required to extend applications with APDL client capabilities. If desired, companies can license the VersatileUX toolkit from Linguistic Technology Systems, Inc., which includes native APDL support.

In general, explicitly designed tools are helpful, but not necessary, for working with for APDL. The following tools can be used or customized for client projects:

1. **RelaeGraph**: RelaeGraph is an in-memory graph library which uses Qt collections. Nodes are wrappers around pointers to user data, whose type must be known to the RelaeGraph system. Graphs are manipulated and queried using C++ operators, which support both triple-style and quad-style annotation policies. RelaeGraphs are intermediate data structures that can be initialized from text files (particularly with a tool called RelaeParser) or built piecewise from C++ applications, and then converted to triples or quads for graph/RDF database back ends.

2. **RelaeParser**: RelaeParser (from "Regular Expression Labeled Annotation Engine") is a Grammar Engine designed to create RelaeGraphs. RelaeParser grammars are more powerful and context-sensitive than most formal grammars, especially BNF grammars. Technically, RelaeParser grammars combine Regular Expressions with function pointers, canonically lambda expression which capture references to RelaeGraphs and contextual information, using RelaeGraph operators to modify graphs in response to Regular Expression matches. Contexts and flags can be active/set or inactive/cleared, modifying which grammar rules are in effect at a given time or context. RelaeParser also extends the default C++ Regular Expression language with several

helpful shortcuts. Grammars are ordinary C++ code; unlike tools such as YACC, there is no external language or precompilation step required to use RelaeParser in C++/Qt applications.

RelaeParser can be used to create project-specific languages for initializing graphs — or any other data structure which can be feasibly expressed as a graph or tree. Despite the ubiquity of graph/tree structures and variety of languages to build and query them, most structures have high-level patterns that are conducive to higher-level grammars. For example, most if not all programming languages compile to Abstract Semantic Graphs, which can in principle be saved, restored, and analyed using a Graph Expresion Language. It would be impractical to write high-level code in a Graph Expresion Language directly, however — imagine, for example, trying to create source-code explicitly in Turtle or N-Quads to directly generate RDF triples, without the aid of source parsers — for similar reasons that it is impractical to program directly in a virtual assembly language, or in XML. RelaeGraph and RelaeParser are innovative in that they introduce notions of a compiler stack into the overall graph database and Semantic Web ecosystem.

3. **QWhite, Q4Graph, and QuadCarve**: These three components are Qt front-ends to graph database systems. QWhite targets WhiteDB, and Q4Graph targets e4Graph; both allow graph data to be stored from Qt objects, and simplify access to the database API. QuadCarve is a more general solution that delegates the actual graph finalization (metaphorically, "carving" data into the store) either to direct wrappers like QWhite and Q4Graph or, to out-of-process systems like AllegroGraph, via HTTP. These three databases represent a cross-section of graph storage engines varying in scale, design, and architecture; on this basis QuadCarve may be augmented with wrappers to other databases whose usage will generally be similar to one of these points on the spectrum. In the case of QWhite and Q4Graph, both are in-memory data stores with the possibilitty of persistance to disk, and can be used to provide long-term data storage for Qt applications. AllegroGraph, on the other hand, is a large-scale commercial platform which can synthesize data generated by many applications, enabling widely distributed application networks. On both ends of the spectrum, APDL can help document serialized data structures and serialization protocols, promoting multi-application data sharing and workflow integration.

4. **DynamoIR**: The "DyNaMo Intermediate Representation" library is a Lisp and Qt/C++ component which can initialize graph structures from special text files. Unlike using RelaeParser, which converts arbitrary source files governed by user-defined grammars, DynamoIR code has a specific format. Technically, DynamoIR can be any Common Lisp code which uses a specific set of FFI (Foreign Function Interface) functions to initialize data via Qt callbacks. DynamoIR is designed to work first and foremost with ECL (Embeddable Common Lisp) and to be called from Qt applications. DynamoIR can work in conjunction with RelaeParser: after assembling RelaeGraphs a compiler engine can generate DynamoIR which (immediately or at some point in the future) can then be evaluated to produce other graphs. That is, DynamoIR can be used to perform transforms between different kinds of graphs even though it is not structurally a Graph Transformation language. Via DynamoIR, RelaeParser becomes a potential compiler front-end with DynamoIR as an (intemediate) compiler target. While DynamoIR can be applied for various contexts, it is particularly designed to produce graphs related to program and source code analysis — including generating APDL descriptions, particularly in the context of a Public Service Interface.

5. VersatileUX: This product is a Qt-based application development toolkit whose libraries provide both data-sharing and customizable GUI classes. VersatileUX can help with both front-end design and multi-application data sharing, as well as the implementation details of populating GUI front-ends with data received from other applications, whether locally or remotely. VersatileUX works natively with APDL and other technologies described here.

VersatileUX includes code in several areas, encompassing cloud sharing, code representation, and workflow management as well as GUI classes. VersatileUX incorporates other components presented here, including DynamoIR, RelaeParser, and QuadCarve, but whereas these latter components are primarily invisible data conduits, VersatileUX helps developers connect these behind-the-scenes technologies to the GUI front-ends which end-users experience directly. For more information, please contact Linguistic Technology Systems, Inc. to read papers or presentations with additional technical details and visual overviews of VersatileUX and its components.

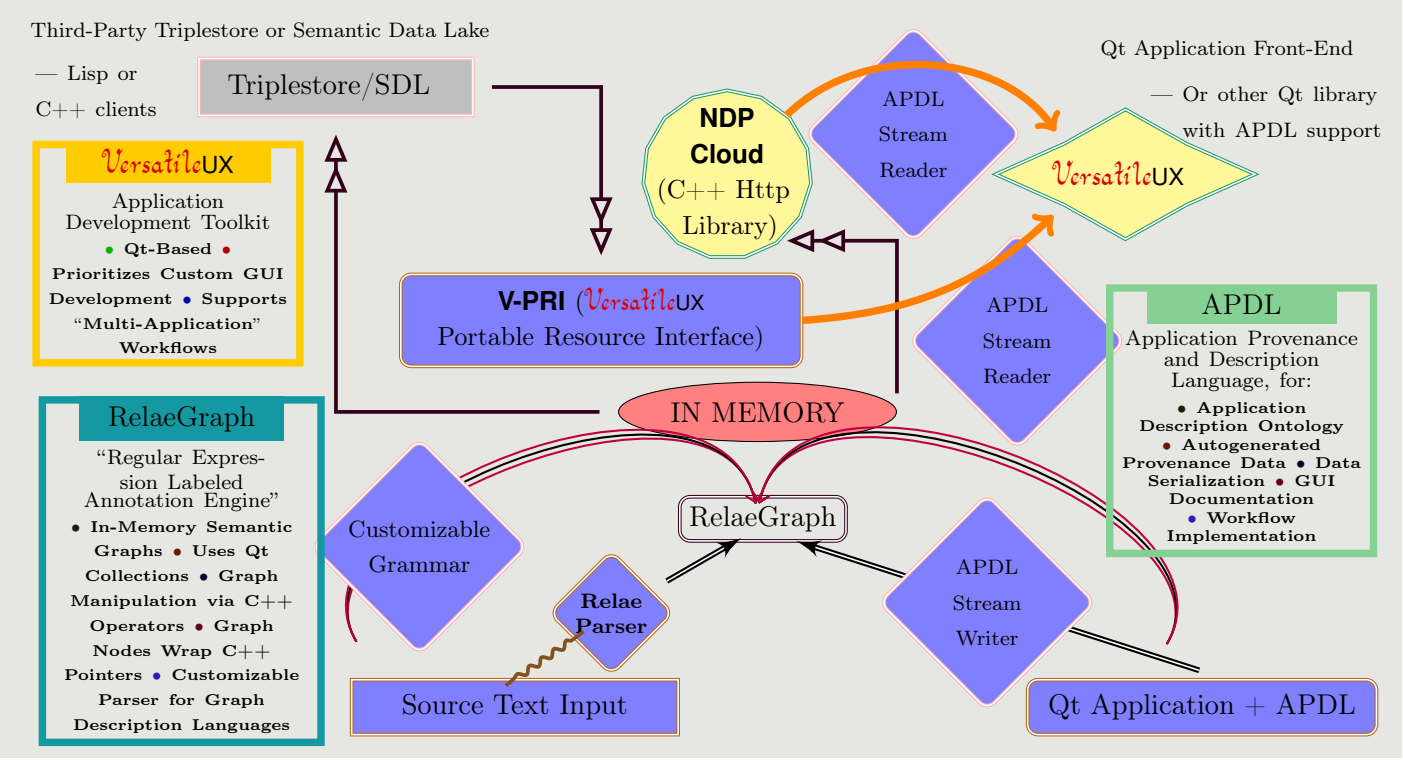

Figure 1: **Outline of APDLTools**

IV. For More Information

APDL is being standardized by Linguistic Technology Systems, Inc., based on studies of organizations' curation of application-related information. For example, as a pilot project, an upcoming book on CyberPhysical Systems, published by Elsevier, will include application metadata along with embedded data sets and code samples to illustrate programming concepts interactively. Parallel initiatives are being pursued in various disciplines, ranging from Real Estate to Bioinformatics. Please contact Linguistic Technology Systems, Inc., for details on these projects, technical documentation, downloads, or licensing information.

Linguistic Technology Systems, Inc.
Amy Neustein, Ph.D.,
Founder and CEO
amy.neustein@verizon.net
201-224-5096