BRIEFINGS IN BIOINFORMATICS. VOL 15. NO 4. 626–636
Advance Access published on 11 December 2012 doi:10.1093/bib/bbs074

Architecture for interoperable software in biology

James Christopher Bare and Nitin S. Baliga

Submitted: 28th August 2011; Received (in revised form): 11th October 2011

Abstract

Understanding biological complexity demands a combination of high-throughput data and interdisciplinary skills. One way to bring to bear the necessary combination of data types and expertise is by encapsulating domain knowledge in software and composing that software to create a customized data analysis environment. To this end, simple flexible strategies are needed for interconnecting heterogeneous software tools and enabling data exchange between them. Drawing on our own work and that of others, we present several strategies for interoperability and their consequences, in particular, a set of simple data structures—list, matrix, network, table and tuple—that have proven sufficient to achieve a high degree of interoperability. We provide a few guidelines for the development of future software that will function as part of an interoperable community of software tools for biological data analysis and visualization.

Keywords: *interoperability; software engineering; bioinformatics; integration; systems biology; data analysis*

INTRODUCTION

Developers of computational biology software increasingly find their programs functioning as parts of data analysis workflows encompassing a number of heterogeneous tools and different data types. Interoperability between these tools is an important and difficult problem, closely intertwined with the well-known and persistent problem of biological data integration [1, 2]. Building the computing infrastructure to seamlessly navigate and recombine biological data remains a work in progress. But some valuable lessons and perhaps a few general principles can be extracted from progress thus far.

The complexity of biological data arises from the diversity of data types (e.g. genotypes, mRNA/protein levels, protein interactions, epigenetic changes and phenotypes), different platforms for measuring/analyzing the same property (e.g. protein interactions), varying quality of measurements and incompatible systems of identifiers (e.g. gene names). Another complicating factor is the need to capture the relevant metadata describing the context of a sample or an experiment, which is needed to make the transition from a mass of individual experiments to a coherent body of data that can be mined for knowledge. All of these data-associated challenges propagate to software engineering where one-off software is essential to support *ad hoc* exploratory data analysis; the challenge in integration arises from the diversity of such software tools that eventually need to be packaged into scripted and repeatable form.

A data-driven approach to biological science depends on collaboration spanning the disciplines of biology, mathematics and statistics, computer science and software engineering [3]. One way to bring about this combination of expertise is to encapsulate domain knowledge in software components, which can be dynamically composed into integrated systems. Such components may be heterogeneous in their choices of languages and components, span all levels of engineering maturity and evolve at

Corresponding author. Nitin S. Baliga, Professor & Director, Institute for Systems Biology, 401 Terry Ave N, Seattle, WA 98109. Tel.: +206 732 1266; Fax: +206 732 1299; E-mail: nbaliga@systemsbiology.org

James Christopher Bare is a software engineer at Sage Bionetworks. He was formerly at the Institute for Systems Biology helping to develop Gaggle, Firegoose, Gaggle Genome Browser and Network Portal.

Nitin S. Baliga is the director of the Institute for Systems Biology, an interdisciplinary institute pioneering the understanding of biological complexity.

different rates. The cutting edge of research will always outpace standardization, generating new data and analysis that may not fit into any existing schema. Federating distributed data sources [4] and independently developed software into an interoperable suite of tools is a challenge that must be addressed in order to build computing systems equal to the task of turning high-throughput data into an understanding of biology in all its complexity.

Our perspective arises through development of software for analysis and visualization of systems biology data [5–7], including early versions of the network visualization tool, Cytoscape [8]. Superimposing gene expression data over Cytoscape networks motivated the development of Gaggle [9], a message passing framework for integration of bioinformatics software. Similar goals motivated several other systems: Galaxy [10], Taverna [11, 12], GenePattern [13], Systems Biology Workbench (SBW) [14] and BioMoby [15]. A common theme that figures prominently into these systems is that of composing separately developed software into suites of tools for the analysis of biological data. Architecting these tools to be interconnected thus becomes a critical step.

We first consider an example data analysis workflow involving several software tools, then present a set of strategies for achieving interoperability. Using these strategies as a means to systematically analyze the interoperability aspects of software architecture provides a few guideposts for the development of future systems.

DATA ANALYSIS WORKFLOWS

Analysis of gene expression (Figure 1) is a common use case that serves as an example for the techniques discussed later. The analysis is divided into steps, having potential for numerous variations and implemented in software that transforms data then passes results onward.

High-throughput measurement of gene expression can be performed by microarray or, increasingly, by sequencing. The shift from arrays to sequencing is an example of technological change that challenges the ability of research software to adapt. In either case, data undergo specialized processing to derive a gene expression matrix, a 2D grid of numeric data in which each row represents a gene's expression profile over changing conditions.

Clustering the resulting matrix is a likely next step, identifying sets of genes with similar expression profiles over the course of the experiment, possibly performed using tools like R [16] or Multi-experiment Viewer (MeV) [17]. Products of co-clustered genes

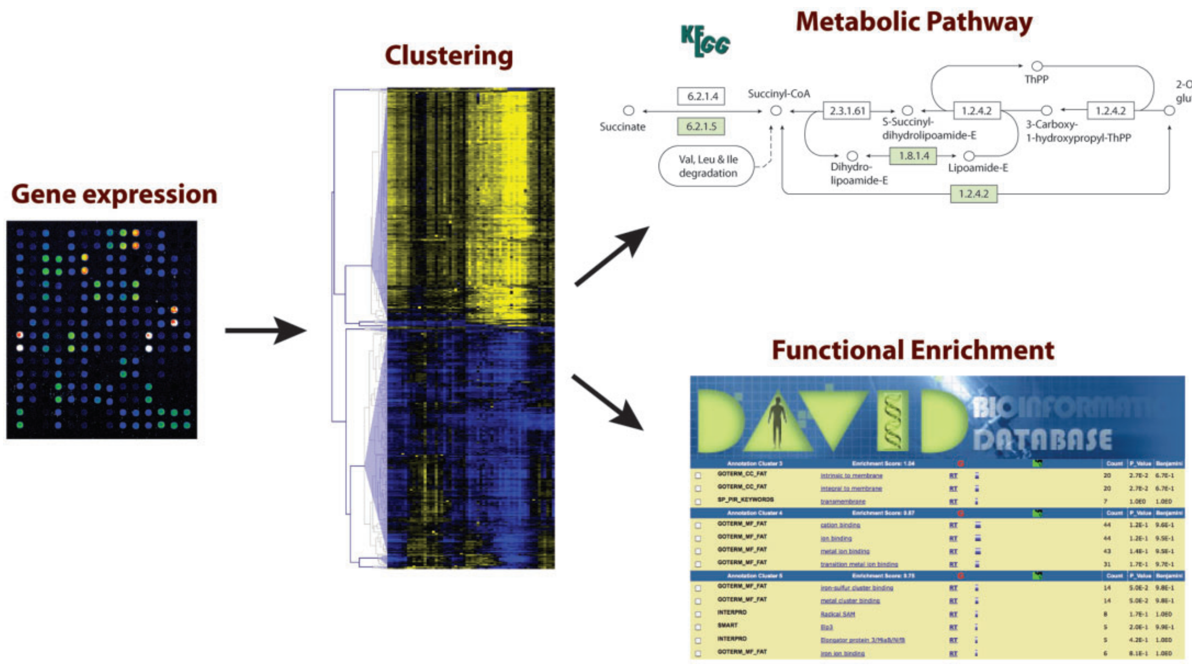

Figure 1: A biological data analysis workflow to cluster and characterize gene expression data. A gene expression matrix derived by microarrays or sequencing experiments is clustered (here we use the data exploration tool MeV) producing lists of co-expressed genes, which are then passed to two web resources for further analysis. KEGG takes gene lists and finds relevant metabolic pathways. DAVID computes functional enrichment.

may have related functions or participate in the same metabolic pathways. The functional annotation tool DAVID [18] accepts lists of genes and computes functional enrichment, returned in tabular form with links to supporting evidence. Through KEGG [19], a list of genes can be submitted as a query returning metabolic pathways represented as a network. Ultimately, the analysis is guided by the design of the experiment, which may seek to connect a disease or environmental stimulus to regulation of specific biological processes. Even this simplified example relies on an impressive array of biological, statistical and algorithmic expertise embedded in interacting software tools.

Similar analyses might run in any of several workflow management systems [20, 21] or be coded into scripting languages. When these tools incorporate mechanisms for packaging and publishing the steps of an analysis, this aids in reproducibility. Like design patterns [22], basic templates for data analysis are independent of particular tools or languages tending to be adapted to fit new situations and reused. Regardless of implementation details, the need for different pieces of software to interact and exchange data underscores the importance of interoperability as a primary concern in the architecture of bioinformatics applications.

STRATEGIES FOR INTEROPERABILITY

A variety of methods have been successfully applied to the problem of building interoperable software systems. APIs, plug-ins, messaging and web services can be seen as variations on the general theme of sharing data and functionality between programs. These strategies overlap and can even, at times, be implemented in terms of one another. Most large software systems employ several of them. They differ in the trade-offs they impose and the degree of separation or sharing between communicating programs and should be selected carefully to yield desired properties.

More complete treatments of these software design strategies are available in the literature on software connectors [23, 24], design patterns [22, 25] and software architecture [26]. We will briefly give enough terminology (Table 1) to discuss a few advantages and consequences in further detail.

Shared representation

Any mutually understandable data format can serve as a shared representation. If two programs can read and write the same file format, access the same database or send intelligible messages to one another, they can communicate. A common format shared by n otherwise unrelated programs means that each program must translate between its internal structures and the shared representation, an n-way translation. This greatly improves on the worst-case scenario where each pair of communicating programs requires its own connector, requiring $n(n-1)/2$ translators to fully connect all programs.

Relational databases are often used as a point of integration. Programs communicating this way will share a dependency on the database schema, but no dependency on each other. In this case, the shared representation is persistent, as are shared files. Messages, as well as objects passed as arguments to

Table 1: Strategies for interoperability

Adapter	A component that translates between incompatible interfaces, protocols or content.
API	Application programming interface; functionality exposed for use by external components.
Broker (mediator or arbitrator)	An intermediary that coordinates interaction between components, serving as the hub in a hub-and-spokes architecture.
Message passing	Sending data from one process to one or more independent processes.
Plug-in architecture	Run-time integration of separately developed task-specific functionality into a general-purpose host program.
RPC	Remote procedure call; a style of interaction characterized by synchronous invocation of specific functionality running in another process.
Shared representation	A commonly understood data format accessed by multiple programs; for example, a shared DB, a common file format (FASTA, GFF, SAM & BAM, RDF and ontologies). A message payload or arguments to an API call can also serve as a shared representation.
Streaming	Processing partial data as it arrives without waiting for a complete transmission.
Web services	An API made available over web protocols (HTTP). SOAP and REST are two common styles.
Workflow	A repeatable pattern of data processing and transformation designed by arranging separate software components to carry out distinct steps.

an API call, are transient but still must be understood by both sides of the communication.

Shared representations might be arranged on a continuum of increasing structure with events and flat text files at one end and relational databases and ontologies on the other, trading off simplicity and generality for precision. Complex schemas and semantically defined vocabularies work well where basic concepts have reached some degree of stability, but this is not always a given in research. The degree to which interoperability can be reduced to a syntactic issue rather than a semantic one deserves consideration, as semantically rich formats come at substantial costs in terms of engineering effort, consensus building and learning curves.

Toward the more structured end of this spectrum are Semantic Web formats including Gene Ontology [27] and BioPAX [28]. The goal of the Semantic Web is to construct universal shared representations, enabling linked structured data to be reused and recombined automatically across application and organizational boundaries [29]. For now at least, Semantic Web technologies must coexist and interoperate with structured data in databases, semi-structured data in various flavors and with unstructured data. Ideally, bioinformatics software will accommodate varying degrees of structure, exploiting semantically rich data where available without excluding lower levels of the structure hierarchy.

Plug-in architecture

Plug-ins are a way to augment a general-purpose tool with specialized functionality without cluttering up the core with functionality pertinent only to a few users. Dependency is one way, with the host having no dependency on its plug-ins. Plug-ins can be distributed separately and are often contributed by third parties. Preserving consistent behavior of an API while core functionality is undergoing rapid development can be challenging. But the host is free to change its internals, as long as the contract of the API is maintained.

The network visualization software Cytoscape has a vibrant community of plug-in developers [30]. Plug-ins can access and manipulate the central data structure, the network, which is maintained by the host program. An upcoming version of Cytoscape is based on the OSGi (http://osgi.org) framework, as is the integrated development environment, Eclipse. OSGi takes plug-in architecture a step further by constructing whole software systems from assemblies of plug-ins. Such exceptionally customizable and reconfigurable tools are well matched to the rapidly evolving complexity of scientific data [31].

Like a plug-in API, an embedded scripting environment is an extension mechanism enabling programs to be augmented with new functionality [32]. For example the editor, Emacs has a Lisp interpreter at its core. The majority of its text editing functionality is written in this language, as are numerous extensions. Mozilla Firefox also follows this pattern with an embedded JavaScript interpreter for executing third-party code and custom extensions, such as Firegoose.

Messaging

Message passing loosely couples independent applications running in different processes by exchanging packages of data in a mutually intelligible format. One common messaging pattern is the remote procedure call (RPC) where a request invoking a specific function is answered by a response, but there are many alternatives [25]. The asynchronous publish-and-subscribe pattern, for example, is often used to propagate events indicating state change or user interaction.

Message-oriented middleware offers a number of attractive features, but at a cost of increasing complexity. Transactional message queues provide asynchrony and buffering and can be configured to guarantee delivery and message order. Message brokers add sophisticated routing and transformations. While powerful, managing this complicated software infrastructure can quickly become prohibitive to all but experts.

Messaging is not well suited to very large data objects. Serialization, copying and deserialization all become expensive in terms of both performance and memory as data size grows, perhaps beyond a limit of a few tens of megabytes with present technologies. Streaming can eliminate the need for whole documents to be in memory, but sacrifices a degree of simplicity. Alternatively, messages may carry a reference to data rather than the full data itself. The pointer can take the form of a URL or a reference to a shared database. This is efficient, but raises again the problem of shared representation, particularly with the introduction of a dependency on a database schema.

Both Gaggle and SBW are message passing systems with binary protocols. The choice of binary versus textual protocols is a trade-off of efficiency against simplicity.

Pipes and filters

Pipes-and-filters is an archetypal technique for interoperability. The pipes-and-filters model [26], often described as 'small pieces, loosely joined', enables workflows to be built up from small programs chained together by streaming. The programs are usually invoked from the command line within a Unix shell and have a single purpose whose behavior is modified by command line switches. Branching is possible, but the tendency is toward linear pipelines. The shared representation is typically text files processed line-by-line.

Web services

Web services and workflows built on them are widely deployed in biology. Building on web protocols brings many advantages from an interoperability perspective. The uniform interface of Hypertext Transport Protocol (HTTP) connects heterogeneous clients and servers. Text representations such as XML or JSON provide platform neutrality.

Web services are often built alongside browser-based web interfaces and, in modern applications, rich web or desktop clients are built on top of well-defined web service APIs. This layered style of architecture supports both interactive and automated access, serving developers, point-and-click users and scripting-enabled power-users.

A request across the network is a relatively slow operation. The granularity of requests should be selected to keep data sizes reasonable and the number of requests small. A pattern is emerging of hosting shared data resources and running analysis on scalable cloud computing infrastructure accessible through web service APIs.

RESTful JSON-based APIs are becoming standard practice and the value of scripting and interconnecting such services is well known. Facebook's OpenGraph, for example, is a platform that supports a vibrant ecosystem of third-party apps. OpenGraph is a web service API with many aspects of a plug-in architecture having the social graph as its central data structure. Similar techniques may cross over easily into the biology domain where the interactions of genes, proteins and species also form densely interconnected networks. Repositories of biological data could similarly act as platforms for integration [33], able to plug in customized modules encapsulating analysis algorithms, visualizations and connectors to multiple data sources.

REST, or representational state transfer [34], is the set of architectural principles underlying HTTP, which incorporates several interoperability strategies. HTTP is an RPC client-server messaging protocol. The client makes requests containing one of a small set of methods, (GET, PUT, POST and DELETE) which are answered by responses from the server. This uniform interface is understood by all web servers. The body of an HTTP message may contain HTML or any of dozens of standard media (MIME) types, including specific formats for images, audio and video. Registration of new media types is an important extension point. Since clients cannot reasonably be expected to support an open-ended variety of media types, browsers offer plug-in mechanisms through which specialized or proprietary media formats can be supported.

One tenet of REST is that information be represented in transit as self-describing messages. The syntax for these documents is typically XML or JSON. Their semantic content and structure may conform to a standard (e.g. HTML) or be application specific. In the next section, we propose a highly general representation of scientific data based on a handful of simple data structures annotated with descriptive metadata.

INTEROPERABLE DATA

Data representation is essential to any method of interoperability, serving as an intermediary between communicating programs with different internal representations. Like the type system for a programming language, an intermediate representation seeks to balance several qualities, among them simplicity, expressivity, universality and efficiency. Our experience with Gaggle suggests a system of interlocking data structures, free of domain specific semantics and general enough to cover a broad range of applications.

These shapes of data—lists, matrices, networks, tables and tuples (nested key/value pairs)—are universal and a capable of representing a variety of biological data types (Figure 2). They are redundant in the sense that it is possible to represent the same data in a number of ways, but a given biological data type usually fits naturally into one of these structures and will be represented by similar structures in the internals of many software tools. Of course, there

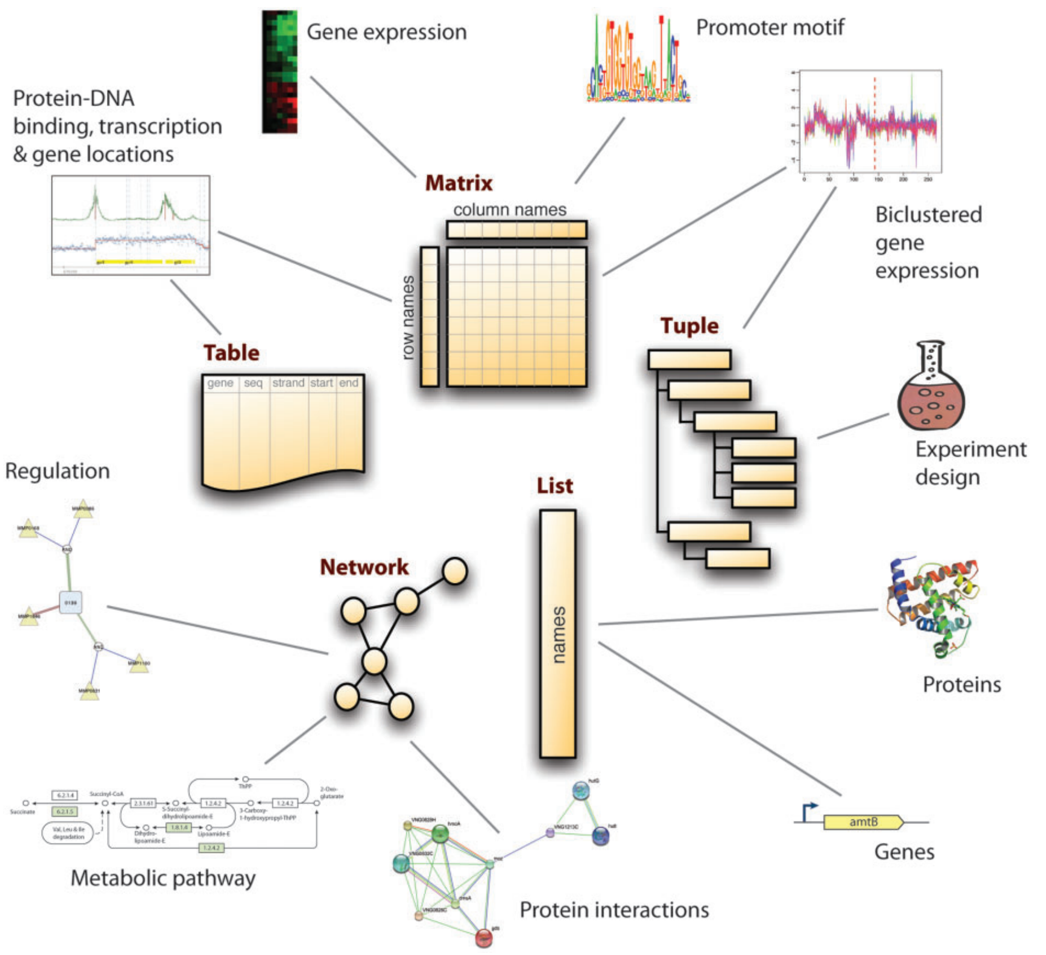

Figure 2: The shapes of scientific data. A wide variety of scientific data can be represented by a handful of fundamental data structures. A list might hold protein or gene identifiers. Networks represent regulatory influence, metabolic pathways or protein interactions. Numeric data resides in matrices, for example a gene expression matrix or promoter motif PSSM. The combination of tabular data and matrices could enable ChIP-chip data, tiling array data and genome features to be plotted by location in the genome. A bicluster, a set of genes co-expressed under specific conditions, might be represented by the combination of a list of genes, a list of conditions and a gene expression matrix, tied together in a tuple (hierarchically nested key-value pairs). Tuples may also represent experiment design (metadata about media, environmental variables or patient data).

are data types that do not fit well into these data structures. Images and sequence, for example, are well served by existing formats and these can be used by reference, as is done in HTTP. As a shared representation, this handful of fundamental data types is sufficient to achieve a surprisingly high degree of interoperability.

Rather than matching complex biological data with equally complex and therefore cumbersome, data standards, we instead control complexity with flexible, generic and composable data types that are purposefully underspecified, letting context fill the gap. In our experience, this strategy works remarkably well at representing biological data, containing the costs in software complexity, remaining amenable to formality where necessary without enforcing it where it is not.

The shapes of data

A handful of data structures sufficient to represent a variety of biological data are:

- **Lists**. A list of identifiers is a basic and universal data structure, which might hold gene or protein names, accession numbers, ontology terms or resource URLs.
- **Matrices**. Libraries for manipulating matrices are fundamental to numerical computation and comprise decades of work, underpinning software like R, MATLAB, and NumPy. In Gaggle, a matrix is

a 2-dimensional array of floating point values with labeled rows and columns, although it might be argued that an n-dimensional array would be a better choice. Gene expression, protein abundance and motifs PSSMs can be expressed as matrices.

- **Networks**. A network, or graph, has nodes connected by edges. Both nodes and edges can have key/value properties attached to them. Protein–protein interaction, gene regulation and metabolic pathways are commonly represented as networks.
- **Tables**. The basic unit of relational databases is the table, a set of rows conforming to a schema. Tables differ from matrices in that each column in a table may be a distinct type of data, for example numeric, string, or boolean. Often, the first column is an identifier and other columns hold categorical or numerical data pertaining to the identified entity. For example, a gene feature table might have columns for gene name, strand, start and end position, and function.
- **Tuples**. Sets of key/value pairs are tuples. The keys are strings that label the values. Simple values can be numeric, string or boolean. Values can also be compound objects: lists, matrices, networks, tables, or other tuples. This nesting enables composition and can be used to build up complex data structures, something that should be done with restraint because it creates a dependency on the precise structure. Like XML and JSON, tuples can represent hierarchical data and can also accommodate RDF triples.

These data structures avoid specifying biological semantics and are to be interpreted in the context of the receiving application, a concept called semantic flexibility. Semantic concerns are left pragmatically in the hands of the user. These go hand-in-hand with the design of the data analysis workflow and of the experiment itself, activities that are likely to remain largely in human hands for some time to come.

Joining data

Joining together corroborating lines of evidence enables robust conclusions. An important aspect of this system of data structures is that they can be readily joined together. For example, a list of genes might be used to select rows in a gene expression matrix or nodes in a protein interaction network. Properties pertaining to those genes might be stored in a table or tuple, also keyed by gene name.

Heterogeneous data sets can be related to each other by joining or merging based on common keys. These common keys, or touch points [35], take many forms in biological data including gene or protein identifiers, ontology terms and loci. Genome browsers, for example, render visualizations by joining data on the basis of location on the genome.

Inconsistent identifiers impede join operations. In spite of several tools [18, 36, 37] for mapping between different naming systems, translating identifiers remains a common source of frustration for bioinformatics researchers. Semantic web technologies, including Life Science Identifiers [38], seek to create a systematic and universal naming system through the use of Uniform Resource Identifiers (URIs). URIs provide a distributed hierarchical namespace thus avoiding naming conflicts, but do not entirely resolve the issues of multiple names for equivalent entities or semantic mapping across related concepts.

The general idea of joining on a common key might be expanded to include more sophisticated mappings. Sequence similarity links together genes across species enabling propagation of information over the phylogenetic tree. A newly sequenced genome can be essentially joined to the body of existing biological knowledge through BLAST. Likewise, functional enrichment connects sets of genes up the hierarchy to biological processes and metabolic pathways.

Interoperability example

Returning to our example gene expression analysis, consider the shapes of data crossing the junctions between the software tools. Aligned sequence reads in tabular format or probe intensities are processed into a gene expression matrix. The matrix is transferred to a statistical tool for clustering. Co-clustered genes, as lists of identifiers, may then be intersected with networks denoting protein–protein interactions, metabolic pathways or regulatory networks returning subnetworks. Gene lists may serve as queries to functional databases returning key-value pairs associating a gene with its function. In this light, composing software to perform successive levels of analysis is largely a matter of sharing these fundamental data structures.

GAGGLE: EXPERIENCE AND LESSONS

Much of our experience putting into practice these strategies for interoperability comes from the development and application of Gaggle, a framework designed for interactive exploratory analysis of biological data. Gaggle integrates several in-house and third-party software tools and has been applied in numerous studies, for example [5–7].

In terms of interoperability strategies, Gaggle is a message passing system. Shared representation takes the form of the set of data structures discussed earlier, with the omission of tables. Messages are propagated through the Boss, a simple message broker that tracks connected applications and routes messages from one to another. Connections are implemented over Java RMI, implying synchronous RPC with binary serialization of data objects. In client applications, Gaggle connectivity is often implemented through plug-in APIs, as it is in Cytoscape. In order to incorporate web applications into the system, an adapter was needed that could translate from the Java RMI protocol to web protocols, HTTP and XML-based web services. This is the basic function of Firegoose.

The Gaggle framework creates a fluid environment for interactive exploratory analysis of systems biology data, integrating an extensible suite of software tools: MeV [17], a graphical tool for clustering, classification and visualization; R and Bioconductor [39]; Cytoscape. Several more tools were developed specifically for use within the Gaggle framework, including a data repository incorporating machine-readable descriptions of experiment design; a translator for mapping across identifier systems and a genome browser [40]. Firegoose [41] connects web applications to the Gaggle framework, exchanging data in either direction between desktop tools and popular web sites such as KEGG [19], STRING [42, 43] and DAVID [18].

As we gained experience with Gaggle, we discovered aspects which worked well and those where greater flexibility and extensibility were needed. Reviewing some of these design decisions may provide guidance for future work.

Tables

Originally, we felt that tables could be adequately represented as matrices or tuples and did not include them in the Gaggle. Of particular concern was the need for communicating programs to agree on the specific contents of the table (the schema). On the other hand, the ubiquity of tables in both spreadsheet-like programs and databases argued for their inclusion. More importantly, significant utility can be had from tables without limiting applications to a prescribed schema. For example, the first column often holds an identifier field. That alone is sufficient for making selections and joining to other data structures. Applications with specific needs might have to look for expected column headers; for example, sequence, strand, start and end indicating a locus on a genome. A program encountering a table with an unfamiliar schema can ignore it, harmlessly. This demonstrates the surprising extent of what can be done without limiting flexibility.

Thus convinced in favor of tables, we faced another problem. Due to the statically typed nature of RMI, Java's binary protocol for remote method invocation, adding this new data type meant recompiling clients, many of which were developed by third parties. Searching for ways to support both older and newer clients, we prototyped a JSON protocol. Easily enabling additive changes, our experimental protocol showed benefits in generality, extensibility and platform independence at some cost of efficiency.

Composition

The original Gaggle included a data type for bicluster—a set of genes co-expressed under a set of experimental conditions. The need for this less general data type demonstrated a fundamental omission. A bicluster is just a pair of lists, a list of genes coupled with a list of conditions. This type of composition was made possible by allowing data objects to be nested inside tuples. A bicluster is now represented as a tuple with two keys, 'genes' and 'conditions', each associated with a list of identifiers. As an added benefit, data objects can now be annotated with metadata. Specifying identifier types or units are among many potential uses.

Microformats

The original Gaggle worked well for desktop applications; however, many useful and popular bioinformatics resources are available as web applications. This motivated the creation of Firegoose, a browser extension that integrates web resources into the Gaggle framework. To make this easier, small amounts of structured data can be embedded directly into web pages, a technique called microformats or microdata in the HTML5 standard. Search engines and browser plug-ins like Firegoose can parse and act on this embedded information enabling data-aware features. One compelling use is to link together web interfaces and web services, which are often different representations of the same underlying data. Web resources can then participate in a seamless data analysis environment along with desktop tools.

Scripting

Gaggle is an environment for exploratory analysis, emphasizing the ability to easily move data from one interactive graphical application to another. In this way, Gaggle differs from workflow tools in that the steps of an analysis are determined interactively by the user, rather than being scripted or designed graphically. Gaggle integrates many programs which were not made with scripting in mind, making it difficult to capture and save analysis workflows. Hooks for scripting graphical applications, perhaps even a shared vocabulary of commands, would help ease the transition from interactive exploratory analysis to automated reproducible workflows.

CONCLUSION

'A "cyberinfrastructure" is a combination of databases, network protocols and computational services that brings people, information and computational tools together to perform science in this information-driven world.'—Lincoln Stein [44]

In his landmark dissertation [34], Roy Fielding systematically described the architectural principles that enabled the success and ubiquity of the web. Can we formulate similar principles that do the same for bioinformatics? Rather than enforcing restrictive and limiting mandates on data and programming models, the necessary flexibility and interoperability might be achieved through a set of general principles and standard practices, much like it has on the web itself.

Requirements

The requirements are relatively clear. The structure of biological research dictates that bioinformatics software be created through a process of distributed independent development with little central coordination. Projects at varying levels of maturity must interoperate while evolving independently at varying rates. Unpredictable new requirements are to be expected. Care should be taken to avoid constraining biological semantics or imposing data models, which may become outdated with advancing biological knowledge.

Bioinformatics software development should remain accessible to scientists and domain experts who are not primarily software engineers. In that spirit, preferred architectures would not restrict choice of programming environment nor require sophisticated development tools or techniques, keeping the barrier to entry as low as possible. *Ad hoc* scripts should be supported and allowed to evolve toward greater engineering rigor as warranted. The goal should be tools and practices that enable both exploratory and repeatable data analysis, cultivate collaborative development and produce open easily exchanged data. These are ideals, but taken as guiding principles, they point to engineering decisions that value flexibility and simplicity.

Principles

'Rule of Composition: Design programs to be connected to other programs.'—Eric S. Raymond [45]

These requirements and values serve to guide the tradeoffs inherent in building complex scientific software. The basic engineering tools for dealing with complexity are abstraction and modularity. Encapsulating specialized functionality behind well-defined interfaces leads to self-contained components. Composition of autonomous components provides the flexibility needed to adapt to unanticipated situations.

Loosely coupled systems are created by carefully limiting dependencies between components. Exchanging data in self-describing documents lowers dependency between components, compared with explicitly invoking another program's functionality. For example, HTTP constrains the set of actions to a handful of operations (GET, PUT, POST and DELETE), pushing almost all variation into the message payload, which can be any of dozens of defined media types, including text, html, xml, images, audio and video or customized application-specific types. The result is that heterogeneous clients and servers can exchange an unlimited variety of data types. Effective extension points are difficult to craft, but once found, enable existing software to adapt gracefully to new demands.

Generality is a key feature of interoperable data structures. Expressing biological concepts in the data rather than in its structure enables new concepts to be incorporated and existing concepts to change. Likewise, plug-in architecture isolates application-specific semantics, which may be in flux, from universal concepts that apply broadly and consistently.

In a rapidly developing field of research, semantics are a vector of change and designing to isolate that change pays off.

The principles of abstraction, modularity, composition, loose coupling, simplicity and generality are simply good software engineering. Applied in context, these principles suggest a set of practices that allow interoperability to proceed naturally. Several existing systems and frameworks exemplify some of the necessary ingredients: composing customized workflows of loosely coupled independently developed software components; protecting against change through generality; promoting extensible or optional standards and keeping software components, protocols and data representations as simple as possible, bearing in mind that specifications that burden developers and data providers with up-front costs are less likely to be adopted. Software tools should be designed for interoperability, anticipating their role as parts of an integrated platform collectively supporting the emergence of yet higher levels as new biology is discovered.

Key Points

- Interoperability is a key feature for scientific software.
- Flexible and powerful software environments for scientific data analysis depend on composition of independently developed software tools each encapsulating domain expertise from particular areas of specialty.
- A handful of simple generic data structures—lists, matrices, networks, tables and tuples (nested key/value pairs)—are capable of representing a variety of biological data types. These represent the shapes of scientific data and provide a basis for simple and flexible interoperability.

Acknowledgements
The authors would like to thank Alexander Pico, Christopher Plaisier, Hector Rovira, Mike Smoot and Wei-ju Wu for insightful discussions. We also thank the reviewers for their helpful comments and suggestions.

FUNDING

This work was supported by the U.S. Department of Energy [award nos DE-FG02-07ER64327 and DG-FG02-08ER64685] and by the National Institute of Health [award nos P50GM076547 and 1R01GM077398-01A2] and by the University of Luxembourg. The work conducted by ENIGMA was supported by the Office of Science, Office of Biological and Environmental Research of the U. S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

1. Stein LD. Creating a bioinformatics nation. *Nature* 2002; **417**:119–20.
2. Stein LD. Integrating biological databases. *Nat Rev Genet* 2003;**4**:337–45.
3. Hood L. Systems biology: integrating technology, biology, and computation. *Mech Ageing Dev* 2003;**124**: 9–16.
4. Madhavan J, Jeffery S, Cohen S, *et al*. Web-scale data integration: You can only afford to pay as you go. *Proc CIDR* 2007;342–50.
5. Bonneau R, Facciotti MT, Reiss DJ, *et al*. A predictive model for transcriptional control of physiology in a free living cell. *Cell* 2007;**131**:1354–65.
6. Koide T, Reiss DJ, Bare JC, *et al*. Prevalence of transcription promoters within archaeal operons and coding sequences. *Mol Syst Biol* 2009;**5**:285.
7. Yoon SH, Reiss DJ, Bare JC, *et al*. Parallel evolution of transcriptome architecture during genome reorganization. *Genome Res* 2011;**21**:1892–904.
8. Shannon P, Markiel A, Ozier O, *et al*. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res* 2003;**13**:2498–504.
9. Shannon PT, Reiss DJ, Bonneau R, *et al*. The Gaggle: an open-source software system for integrating bioinformatics software and data sources. *BMC Bioinformatics* 2006; **7**:176.
10. Giardine B, Riemer C, Hardison RC, *et al*. Galaxy: a platform for interactive large-scale genome analysis. *Genome Res* 2005;**15**:1451–5.
11. Oinn T, Addis M, Ferris J, *et al*. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 2004;**20**:3045–54.
12. Hull D, Wolstencroft K, Stevens R, *et al*. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res* 2006;**34**:W729–32.
13. Reich M, Liefeld T, Gould J, *et al*. GenePattern 2.0. *Nat Genet* 2006;**38**:500–1.
14. Hucka M, Finney A, Sauro HM, *et al*. The ERATO Systems Biology Workbench: enabling interaction and exchange between software tools for computational biology. *Pac Symp Biocomput* 2002;450–61.
15. Wilkinson MD, Links M. BioMOBY: an open source biological web services proposal. *Brief Bioinformatics* 2002;**3**: 331–41.
16. Ihaka R, Gentleman R. R: a language for data analysis and graphics. *J Comput Graph Stat* 1996;**5**:299–314.
17. Saeed AI, Sharov V, White J, *et al*. TM4: a free, open-source system for microarray data management and analysis. *Biotechniques* 2003;**34**:374–8.
18. Dennis G, Sherman BT, Hosack DA, *et al*. DAVID: database for annotation, visualization, and integrated discovery. *Genome Biol* 2003;**4**:R60.
19. Kanehisa M, Goto S. KEGG: kyoto encyclopedia of genes and genomes. *Nucleic Acids Res* 2000;**28**:27–30.
20. Li P, Castrillo JI, Velarde G, *et al*. Performing statistical analyses on quantitative data in Taverna workflows: an example using R and maxdBrowse to identify differentially-expressed genes from microarray data. *BMC Bioinformatics* 2008;**9**:334.

21. Kuehn H, Liberzon A, Reich M, *et al*. Using GenePattern for gene expression analysis. *Curr Protoc Bioinformatics* 2008. Chapter 7: Unit 7.12.
22. Gamma E, Helm R, Johnson R, *et al*. *Design Patterns: Elements of Reusable Object-Oriented Software,*. Addison-Wesley Professional, 1995.
23. Mehta NR, Medvidovic N, Phadke S. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.
24. Mehta NR, Medvidovic N, Phadke S. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering,* 2000; 178–87.
25. Hohpe G, Woolf B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.
26. Garlan D, Shaw M. An Introduction to Software Architecture. In: Ambriola V, Tortora G, (eds). *Advances in Software Engineering and Knowledge Engineering,* Vol. I. World Scientific Publishing Company, 1993.
27. Ashburner M, Ball CA, Blake JA, *et al*. Gene Ontology: tool for the unification of biology. *Nat Genet* 2000;**25**:25–9.
28. Demir E, Cary MP, Paley S, *et al*. The BioPAX community standard for pathway data sharing. *Nat Biotechnol* 2010;**28**: 935–42.
29. Lord P, Bechhofer S, Wilkinson M, *et al*. Applying semantic web services to bioinformatics: experiences gained, lessons learnt. Semantic Web—ISWC *2004.* 2004;**3298**:350–64.
30. Killcoyne S, Carter GW, Smith J, *et al*. Cytoscape: a community-based framework for network modeling. *Protein Netw Pathw Anal* 2009;**563**:219–39.
31. Börner K. Plug-and-play macroscopes. *Commun ACM* 2011;**54**:60.
32. Technomancy - in which three programming methods are compared. http://technomancy.us/161. (26 May 2012, date last accessed).
33. Rovira H, Killcoyne S, Shmulevich I, *et al*. An integration architecture designed to deal with the issues of biological scope, scale and complexity. *Data Integr Life Sci* 2010;**6254**: 179–91.
34. Fielding RT. *Architectural Styles and the Design of Network-Based Software Architectures,* 2000.
35. Goble C, Stevens R. State of the nation in data integration for bioinformatics. *J Biomed Inform* 2008;**41**:687–93.
36. van Iersel MP, Pico AR, Kelder T, *et al*. The BridgeDb framework: standardized access to gene, protein and metabolite identifier mapping services. *BMC Bioinformatics* 2010;**11**:5.
37. Smedley D, Haider S, Ballester B, *et al*. BioMart—biological queries made easy. *BMC Genomics* 2009 14;10:22.
38. Clark T, Martin S, Liefeld T. Globally distributed object identification for biological knowledgebases. *Brief Bioinformatics* 2004;**5**:59–70.
39. Gentleman RC, Carey VJ, Bates DM, *et al*. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol* 2004;**5**:R80.
40. Bare JC, Koide T, Reiss DJ, *et al*. Integration and visualization of systems biology data in context of the genome. *BMC Bioinformatics* 2010;**11**:382.
41. Bare JC, Shannon PT, Schmid AK, *et al*. The Firegoose: two-way integration of diverse data from different bioinformatics web resources with desktop applications. *BMC Bioinformatics* 2007;**8**:456.
42. von Mering C, Jensen LJ, Snel B, *et al*. STRING: known and predicted protein-protein associations, integrated and transferred across organisms. *Nucleic Acids Res* 2005;**33**: D433–7.
43. Jensen LJ, Kuhn M, Stark M, *et al*. STRING 8—a global view on proteins and their functional interactions in 630 organisms. *Nucleic Acids Res* 2009;**37**:D412–6.
44. Stein LD. Towards a cyberinfrastructure for the biological sciences: progress, visions and challenges. *Nat Rev Genet* 2008;**9**:678–88.
45. Raymond ES. *The Art of Unix Programming*. Addison-Wesley Professional, 2004.