OMICS A Journal of Integrative Biology
Volume 7, Number 1, 2003
© Mary Ann Liebert, Inc.

Evolving from Bioinformatics in-the-Small to Bioinformatics in-the-Large

D. STOTT PARKER,[1] MICHAEL M. GORLICK,[2] and CHRISTOPHER J. LEE[2]

ABSTRACT

We argue the significance of a fundamental shift in bioinformatics, from in-the-small to in-the-large. Adopting a large-scale perspective is a way to manage the problems endemic to the world of the small—constellations of incompatible tools for which the effort required to assemble an integrated system exceeds the perceived benefit of the integration. Where bioinformatics in-the-small is about data and tools, bioinformatics in-the-large is about metadata and dependencies. Dependencies represent the complexities of large-scale integration, including the requirements and assumptions governing the composition of tools. The popular *make* utility is a very effective system for defining and maintaining simple dependencies, and it offers a number of insights about the essence of bioinformatics in-the-large. Keeping an in-the-large perspective has been very useful to us in large bioinformatics projects. We give two fairly different examples, and extract lessons from them showing how it has helped. These examples both suggest the benefit of explicitly defining and managing *knowledge flows* and *knowledge maps* (which represent metadata regarding types, flows, and dependencies), and also suggest approaches for developing bioinformatics database systems. Generally, we argue that large-scale engineering principles can be successfully adapted from disciplines such as software engineering and data management, and that having an in-the-large perspective will be a key advantage in the next phase of bioinformatics development.

INTRODUCTION

BIOINFORMATICS IS A RAPIDLY DEVELOPING SCIENTIFIC DISCIPLINE in which progress can be charted through three successive stages: *discovery*—determining the basic data types and problems that define the field; *analysis*—solving key problems and identifying tractable subproblems; and *integration*—constructing a unified framework that resolves questions by combining available analyses.

Relying on concrete examples, we outline the challenges of analysis versus integration. Specifically, we identify problems that emerge when integration is lacking, as well as a consistent set of principles that address these problems. Viewing these principles as defining *"bioinformatics in the large,"* we sketch pragmatic approaches for building systems that support large-scale bioinformatics.

Bioinformatics can learn from other fields that have already confronted the challenges of large-scale in-

[1]Computer Science Department, University of California, Los Angeles, Los Angeles, California.
[2]Department of Chemistry and Biochemistry, University of California, Los Angeles, Los Angeles, California.

tegration. For example, software engineering lessons learned from building massive, industrial-strength software systems can be useful in bioinformatics. In their seminal 1976 paper "Programming In the Large Versus Programming In the Small," DeRemer and Kron (1976) distinguished between two scales of software development. *In-the-small* issues generally lie within the comprehension of one person and the jurisdiction of a single craft. Here reside problems regarding algorithms, specific mathematical models, and expertise of an individual scientist. *In-the-large* issues focus on dependencies that span component boundaries and cross jurisdictions. Here reside problems of maintaining connections among different data and disciplines, modeling how tools interact and change, and answering complex queries across all the data.

Bioinformatics also has much to learn much from database engineering, whose focus, from its inception, has been in-the-large. This difference in focus is a key aspect of the well-known mismatch between the capabilities of existing database systems and the needs of bioinformatics. We believe that an in-the-large perspective, adapting large-scale database engineering principles to fit bioinformatics, can help resolve this mismatch.

For many disciplines, the distinction between in-the-small and in-the-large is critical. How does this distinction apply to bioinformatics? Most research in the field today focuses on analysis, reducing individual problems to manageable proportions with specialized tools. Examples include powerful algorithms for genome assembly, clustering of microarray expression data, prediction of three-dimensional protein folding, and tools for sequence homology search and alignment. In each case, the tools are specialized—producing a specific data transformation for a particular type of data (say, mapping an input protein sequence to an output fold prediction). In short, most work today is in-the-small (we emphasize that the term is descriptive, not pejorative)—as analysis is a core activity of bioinformatics.

However, there are many signs that in-the-large considerations may prove important in bioinformatics:

- The enormous diversity and complexity of bioinformatics data. No one person can understand all of the data types, much less look at all of the data.
- The need for strong data integration. Although it is tempting to treat these diverse data types separately, they are strongly interrelated and will need to be integrated.
- The need for automation. Each year there is 10 times more data than the previous year, and any process on this data that is not automated will fail to scale.

The growing importance of these challenges is demonstrated by much ongoing work in the field. Can these problems be solved by conventional in-the-small tools and approaches? We think this shift is not merely a matter of degree (that is, "we just need more tools") but a *qualitative* change in the scope of tools and architecture required.

IN-THE-LARGE CHALLENGES

Our experiences in the "hard knocks school" of bioinformatics data integration illustrate why it is difficult to solve these problems with the typical in-the-small methodologies. Here, we discuss two case studies: *GeneMine*, an interactive data-mining tool for biologists to analyze gene and protein structure-function (Lee and Irizarry, 2001); and our single nucleotide polymorphism (SNP) discovery system (Irizarry et al., 2000), which has produced about a quarter of the total coding region SNPs currently known in the human genome.

Gene Function Analysis

At first glance, in-the-large methodologies are not necessary for 'small' projects such as a single biologist analyzing the function of one gene using bioinformatics tools. However, in our development of *GeneMine* for this task, we found an in-the-large perspective invaluable. *GeneMine* (Lee and Irizarry, 2001) is an interactive environment for biological sequence analysis that supports fully automated functional and feature discovery on DNA and protein sequence data, including: (1) functional and structural features, including functional motifs, secondary structure, predicted fold, domains; (2) homology families, analyzed and cross-validated by family "finger-printing"; (3) expression patterns, indicating tissue, cellular, or disease-specific expression levels, working with data from the dbEST, TIGR, and Incyte LifeSeq databases; and (4) disease/association data, including genetic mapping, polymorphism, and other disease association data.

The *GeneMine* display, shown in Figure 1, includes a *structure window* at upper left (for three-dimensional atomic structure and molecular modeling), *functional annotations* (function features associated with specific residues of sequences), and *the information window* at upper right (for drill-down, browsing, and user hypertext documents containing embedded views of the three previous kinds of data). These views are interdependent: any action in one is reflected in all, permitting users to perceive and explore dependencies among different forms of information. This linking of different models is by itself extremely powerful. It emphasizes similarities and differences between models, and it creates an "aggregate" model from individual ones.

GeneMine relies on numerous in-the-small analytic tools. Their characteristics posed several challenges for its design:

Most Tools are highly specialized

A defining characteristic of in-the-small tools is that they answer one particular query, usually for a specific data type or pair of related data types. For example, BLAST finds relationships within the single data type "sequence." Finding a relationship between a different pair of data type x and y usually requires crafting a new tool for just that pair—even if other tools already relate x and y to other known data types. Typically, each new query requires a new program (or script) to be written.

Browsing is not querying

Even when substantial effort is applied to provide an integrated interface to multiple types of data, in-the-small methodologies leave users with limited query capabilities. By itself, in-the-small tool development does not confer general query or data-mining capability but merely the ability to *browse.* Indeed, a prevailing model

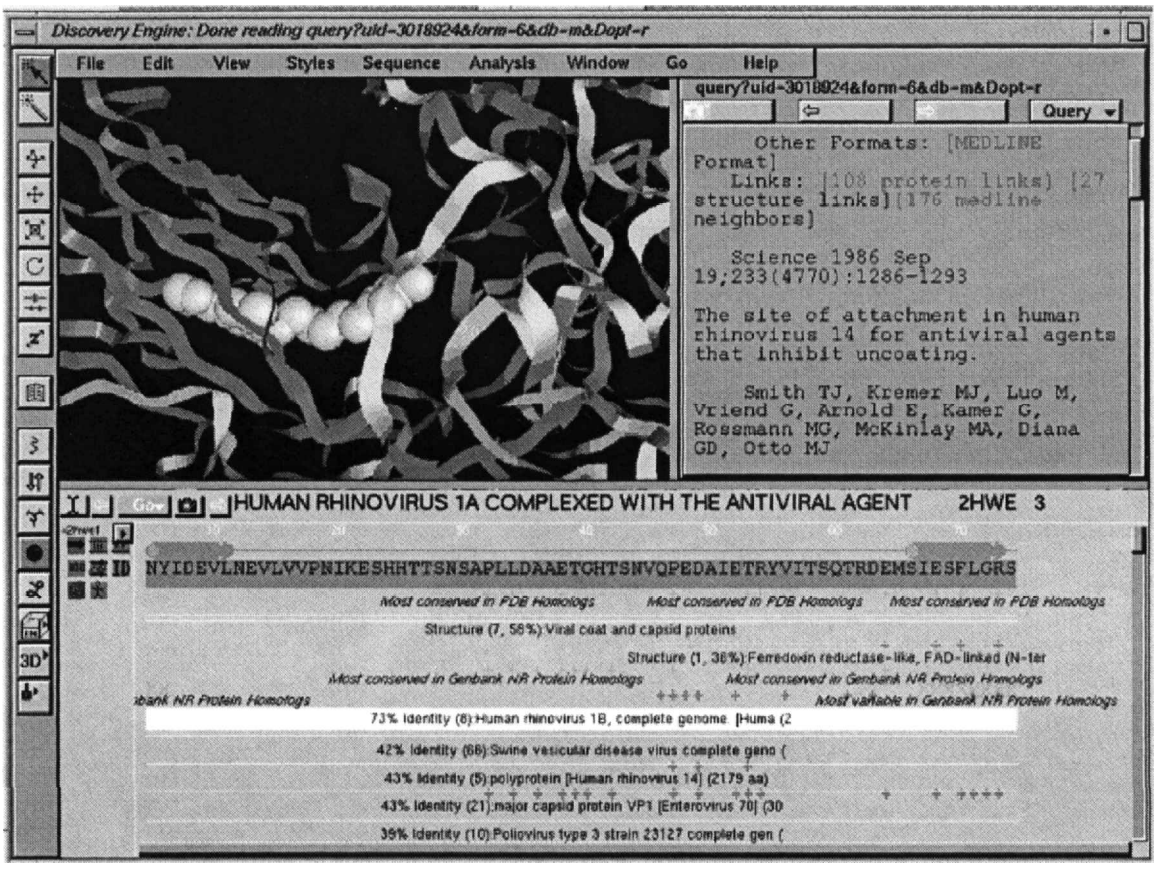

FIG. 1. A view of the GeneMine application (displaying the rhinovirus 1a crystal structure 2HWE [PDB]).

.chromat .phd .phdfas .screen	raw chromatogram data PHRED format PHRED & FASTA combined result of crossmatch	.contigs .fas+cont .fas .reo .qual	sequence contigs contigs & FASTA combined FASTA format reorientations remapped quality data	.po .pir .pocons .snp .allele	partial order alignment graph PIR alignment format consensus sequences SNP scores allele and supporting evidence

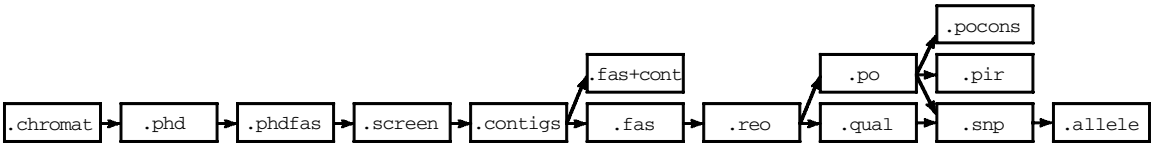

FIG. 2. Dependencies in a Makefile for the SNP discovery pipeline.

for this style of integration is simply a set of linked web pages in which each data type gets its own web page with links to web pages for other data types in bioinformatics.

Expertise is key

A knowledgeable user can follow these links to browse the network of relationships. At each step the user needs expertise to understand how to use that specific tool, how to interpret its results, and what other tools ought to be applied. In general the knowledge for effective use of a network of in-the-small tools resides *entirely with the user.* In the absence of an in-the-large framework capturing "how to use this tool" as metadata, only users can determine the composition of in-the-small tools needed to solve a problem.

Most experts are inexpert at most things

Expertise is narrow by its very nature. Generally, researchers are not expert about most of the many data types in bioinformatics. Even if we assume universal expertise, people can still become "lost" when using most web pages and tools (in the sense that they don't understand a specific in-the-small tool adequately). Even worse, most users would not necessarily know that a given tool (solving a particular problem) exists. In the absence of an in-the-large framework for which the construction of such tool chains is automated, individual users must painstakingly compose these chains by hand—a time-intensive and error-prone process.

In-the-small tools alone are an incomplete query model

Because in-the-small tools are specialized, answering a new query usually requires writing a new specialized script. This raises a general problem: with in-the-small development, any query relating pair of distinct data types x and y corresponds to a single script. If there are n data types, then all queries relating pairs among them can require $O(n^2)$ scripts. If only a few static queries are required, this is not a problem. However, in a dynamic research environment where new queries are constantly posed, this becomes a barrier to discovery.

SNP Discovery Pipeline

Our system for detecting SNPs comprises a large number of software components (from many sources), processing many data types, and a large volume of data (approximately 4 million independent EST sequences and chromatograms). A pipeline developed for this task is depicted in Figure 2. Over the course of work on this project (from 1998 on), we came to appreciate the technical challenges of large-scale analyses. Looking back our problems fell into three categories:

Pipelines are powerful but an incomplete query model

The pipeline—and more generally the concept of *knowledge flow* discussed below—is an extremely useful construct in data mining. As Figure 2 suggests, it is natural to view bioinformatics queries as filters on knowledge flows constructed with pipelines. Also, as is well-known, pipelines *scale:* they can handle large volumes of data. However, pipelines by themselves cannot express many useful queries that require more general networks of flows.

Demands of change and stability are at odds

Throughtout the project, neither the data, nor their schemas, nor the software were static; instead, all changed rapidly. While rapid schema evolution might be considered unacceptable in a typical software development or data base project, it is the norm in discovery science. In this context it is essential to maintain dependencies, both among different data types, and over time in individual data types. For example, the same SNP might be identified in independent runs on different dates, and it is important to keep a stable ID for it throughout these runs, even if its relationship to many other data types changes dramatically (*e.g.*, being switched from one gene cluster to another by a better clustering algorithm).

In-the-small development neglects dependencies

Our initial in-the-small methodologies (such as keeping data as flat files within the file system and writing monolithic processing scripts to run the long series for the analysis) exacted a high cost in terms of data management complexity, processing errors, constant rewriting of scripts even for minor changes to the pipeline, and confusion over potentially inconsistent combination of various version of data, tools, and scripts. Fundamentally, in-the-small tools work best in a static context, because they lack a model of how tools evolve, how they interact, and how they can be combined in a general and automatic manner. the critical importance of these considerations, which are common in bioinformatics, forced us to adopt an in-the-large methodology.

BIOINFORMATICS IN-THE-LARGE AS MANAGING KNOWLEDGE MAPS

Whereas in-the-small tools consume user data, in-the-large tools consume *metadata* (data describing user data). To address the many challenges just raised, it can help to adopt an in-the-large perspective. If we characterize in-the-small development as *making roads* relating one data type to another, in-the-large development is *making roadmaps*. More specifically, if in-the-small roads are tools, then in-the-large roadmaps specify dependencies among tools, such as the requirements and assumptions governing their composition. Much as complex navigation requires roadmaps, in-the-large bioinformatics requires *knowledge maps*.

Roughly, by a knowledge map, we mean a repository of metadata, or model, describing types of knowledge, knowledge flows, and dependencies among these. By a knowledge flow, we mean a tool or service that implements some mapping among types. Dependencies include *derivation rules*—specifying how to derive one flow or type from others, and how to propagate changes in one flow or type to others. In database terms, dependencies can correspond to integrity constraints, triggers, indexes, or views.

Knowledge maps admit automation

Bioinformatics in-the-large requires automation. Each year there is 10 times more data than the previous year, and any process of understanding this data that is not automated will fail to scale.

Knowledge maps are important for success in-the-large because they admit multiple forms of automation. First, information can be *automatically derived* from individual knowledge map elements. For example, rather than make explicit the $O(n^2)$ transformations needed to interconnect n data types, it can be sufficient to provide only $O(n)$ transformations: a knowledge map that "understands" the individual data types and the tools interrelating them can automatically derive a sequence of transformations (a chain of dependencies) if such a sequence exists.

Second, much of the value of data integration hinges on helping users move beyond the bounds of their individual expertise. Requiring that the knowledge to use a network of tools rests with the user drastically limits how the tools will be used, and any single user's ability to focus on the science.

Third, automating integrity enforcement permits users to manage dependencies correctly and efficiently, and frees them from detail. They can then elevate their concerns—from the computational equivalent of bookkeeping to scientific exploration, and from scientific method to science.

Knowlege maps embody well-known large-scale systems design principles

The knowledge map is a consequence of well-known large-scale systems design principles, specifically two general principles for scalability:

- *Scalable design*: separate metadata from data (defining a knowledge map) in a way that provides abstraction, encapsulation, and definition of dependencies.
- *Scalable operation*: use metadata to provide automated compilation and optimization of queries (and automated derivation of dependencies), scalable high-performance execution using independent and pipeline parallelism, and enforcement of dependencies.

These principles characterize an *"in-the-large perspective."* They are embodied in large-scale software engineering models such as the UML (See URLs at end), and more generally in *configuration management systems* (Bendix, 1994). They are also embodied in modern database systems.

Knowledge maps in make

Metadata can encode effective models of dependencies: how user data and software tools interact, and even consequences of their evolution.

The Unix tool *make* is an extremely popular tool for managing dependencies among file types. It automatically generates the sequence of steps needed to construct a desired target. Whereas each step is a script implementing a *road*, the *Makefile* is a *roadmap* from which traversals can be derived. The *suffix rules* of *make* also allow one to define file types and general rules for managing instances of these types, an extremely powerful facility.

DeRemer and Kron (1976) emphasized the importance of what they called *module interconnection languages* for moving from in-the-small software development to in-the-large software engineering. Viewing these languages as metadata, they provided a kind of knowledge map.

DeRemer and Kron's work just preceded the advent of *make*, however—and in our opinion *make* has been a more powerful technology than their module interconnection languages for progress towards software engineering in-the-large. We see *make* as a compelling prototype of systems for managing knowledge maps, and a technology for bioinformatics in-the-large. We discuss *make* in more detail below.

Knowledge maps in GeneMine

Biologists are accustomed to thinking in terms of diverse "types" of genomic information (chromosome, gene, mRNA, protein, and the like).

In *GeneMine* these types are modeled in a knowledge map called the *catalog*. The catalog is a graph database, in which each data type is a node, relationships are edges, and prediction tools or search algorithms are indexes that construct graphs dynamically as the user explores a given region of the graph. *GeneMine* required construction of a mapping system to explicitly represent and use the roadmap of all existing "roads" and data types (i.e., edges and nodes of the graph).

One of the key design goals in *GeneMine* was to change the data-mining model from *expert query* to *information push*. In the standard web model, a biologist can answer a given question only if he knows of the right tool, and has the expertise to access, use, and interpret it correctly. By contrast, *GeneMine* automatically runs *all* tools that are bound in the knowledge map to the user's current data types, filters the results for significance, and presents them in a unified data-mining interface of aggregations, pivots, and drill-downs.

Information push leads the user to new connections in a framework designed to make travel through the knowledge map more effortless and instantaneous (with results within seconds). As a direct benefit of this automation, the focus of a user's effort switches from the mechanics of tool selection, composition, and invocation to interpreting the many results that *GeneMine* determines automatically and to the exploration of suggestive interconnections (Lee and Irizarry, 2001).

Knowledge maps in SNP discovery

Figure 2 illustrates how our SNP discovery process was developed to exploit dependency management with *make*. More generally, modeling change was essential for managing the dynamic character of the process, and it has rendered our complex SNP calculation flexible, reliable, and easy to manage and modify. This has been an indispensable key to our ability to produce, at much lower cost, as large a contribution to coding region SNP discovery as the other major SNP discovery efforts (the Human Genome Project, and the independent SNP Consortium).

MANAGEMENT OF KNOWLEDGE MAPS WITH *MAKE*

A concrete starting point and inspiration for bioinformatics in-the-large is the popular Unix tool *make* (Feldman, 1979; Oram and Talbott, 1991). An essential for the distribution and construction of software applications and systems both large and small, many see it (narrowly) as a tool for constructing executable binaries. *make* is a surprisingly good sketch of a system for managing knowledge maps. It provides three in-the-large features:

- Explicit definitions of types and dependencies (derivation rules)
- Dependency abstraction (through enscapsulation and suffix rules)
- Monotonic integrity enforcement (assuming there are no cycles among the dependencies)

Figure 2 shows the suffix rules of a *Makefile* used to implement the SNP discovery pipeline discussed earlier.

Makefiles *as knowledge maps*

Abstractly, *make* rules are dependencies of the form T:D C where target T is a denotation for the information product that *make* will derive, $D = d_1, \ldots, d_k$ is an enumeration of the antecedents on which T is dependent, and C is a sequence of commands that *make* will execute to derive T. Executing "*make* T" enforces "version monotonicity": first, if any of the antecedents d_i do not yet exist, it is recursively made; then, if at least one antecedent is more recent than T, T is made (by executing commands C).

Suffix rules are an important feature of *make* that specify dependencies among types. Following the convention that filename suffixes define the 'type' of information in a file, a suffix rule generally has the form .d.t: C, indicating that any file *f*.t can be derived from file *f*.d by executing commands C. Figure 2 reflects a number of these rules.

While the precise details of *make* are not that important here, some essentials are worth noting:

- Types and dependency rules are given explicitly.
- Suffix rules define knowledge flows among types.
- Rules provide information encapsulation: each rule is self-contained and can be modified largely independently of other rules.

Monotonic integrity enforcement

Monotonicity is an important property whenever dependencies are constraints that are enforced by executing commands. Monotonicity guarantees incremental progress, in which enforcing dependencies can only make things "better" (preserve an integrity ordering regarding dependencies). Monotonicity is a key concept in database systems, in the forms of both transaction management and integrity enforcement. It is a vital in-the-large property that permits local changes with reasonable assurances that the effects of such changes are bounded. With *make*, monotonicity normally requires the dependency rules to be acyclic (Feldman, 1979), and it imposes a versioning discipline under which enforcing a dependency can only make files more up-to-date (Gunter, 2000).

We believe that scientific data mining rests on chains of derivation from which ad-hoc branches may be easily constructed, amended, and maintained. The incremental discipline of development with *make* lends itself to this style of exploration, and it meshes well with accepted practices in source code management and revision control (such as with CVS and Subversion). This discipline encourages users to share *Makefiles* that codify common or useful laboratory practices and analyses, thereby improving productivity and reducing opportunities for error. Explicit combinations of configuration management and version control have been developed recently (Bendix, 1994; Gunter, 2000).

Make *as a query language*

Makefiles can be used to define "queries." The *make* rules T:D C and .d.t: C define different kinds of database *views.* Our experience is that a system like *make* can be very effective as a large-scale query infrastructure. Putting this more generally, *make*-like systems can provide an in-the-large "query language."

Today's database systems do not yet support general *"query configuration management,"* but managing the complexity of bioinformatics requires something like it: keeping track of diverse data sources and representations, chains of derivation, and webs of dependencies, while at the same time enforcing requirements on data integrity, consistency, and quality.

IMPROVING ON *MAKE* FOR MANAGING KNOWLEDGE MAPS

As just illustrated, *make* has many capabilities that make it valuable for bioinformatics, and it neatly captures differences between bioinformatics in-the-large and bioinformatics in-the-small. However, it was never intended as infrastructure for bioinformatics, and there are a number of ways to improve on *make* for managing knowledge maps.

Managing scientific dependencies

Bioinformatics, like all sciences, has its own culture of integrity and rigor, and its own conventions for managing information dependencies. *make*'s useful version monotonicity property can be generalized to capture these notions of dependencies and integrity.

Reproducibility of results, for example, is a foundation of scientific integrity. While large-scale computation has made it possible to obtain new results, it also has made it impossible to reproduce some of these—specifically those whose provenance was not formally recorded. Tracking this provenance is difficult—as a result of the same complexity raised earlier—a data product may be the end result of a long chain of derivation, which in turn may rely on numerous other data sources having their own history and lineage. Worse, a blizzard of tools and scripts may be invoked along the way. Reproducing a data product can require reproducing a complex ecology of data sources, tools, operating systems, and hosts.

Maintaining explicit dependency relations would address this problem, and *make*'s dependencies give a step toward this. When managed with modern source code revision tools such as CVS or Subversion (see URLs at the end), a *Makefile* can be an archival record of derivation of an information product. We can anticipate the development of ancillary tools, for example, to describe the differences in the provenance of multiple versions of the same information product—much as source code revision systems can summarize the differences among versions of source files for a software product.

Similarly, just as application developers request "the latest stable version" of a software system, or "the most recent experimental branch that passed the regression tests," researchers could, with appropriate tool support, demand an analysis that "relies on the most reliable data" or a derived product which incorporates "the latest microarray run." In short, once made explicit, provenance could be examined, manipulated, and parameterized.

Implementing knowledge maps with extensible configuration management

Today, there are many configuration management tools and integrated development environments that improve on *make*. While powerful, *make* has a number of limitations. For example, in developing the SNP pipeline, the limitation of single-input, single-output suffix rules was a problem, since the final SNP scores depend on multiple inputs (Fig. 2). Each descendant of *make* addresses specific limitations.

For applicability of a configuration management system in bioinformatics, extensibility seems vital. Several efforts to improve on *make* are integrated with a programming environment, such as *Cook* (Miller, 2002) and *SCons* (Knight, 2001). A promising approach is to adopt a scripting language as the foundation for this combination. Candidates include *Perl* and *Python.*

For managing knowledge maps, programming environments are easily augmented with support for graphs. Powerful graph programming frameworks such GVF (Marshall et al., 2003), GTL, GFC, and LEDA (see URLs at the end) illustrate how graph capabilities can be added easily to an existing language. These capabilities are adequate for implementing basic operations on knowledge maps such as deriving paths between types, or searching for types with certain properties.

Scripting languages can also provide support for queries in a way that is consistent with the notion of knowledge flow. A coarse-grain, large-scale query language like *make* and a medium-grain traversal mech-

anism for graphs can blend with finer-grain query and transformation primitives like those provided by *Kleisli* (Wong, 2000) and *Python* (van Rossum and Drake, 2001). At this scale, *comprehensions* (Trinder, 1990) are a popular query construct, providing "set notation"–like syntax. The *Python* comprehension *[g(x) for x in C if p(x)]* yields the collection of values *g(x)* obtained from *x* in collection *C* that satisfy the predicate *p(x)*, for example. In particular, *Python* implements eager list comprehensions and lazy list comprehensions, as generators (Schemenauer, 2001). Like containers, the latter are flows. *Python* also implements functional notions such as lambda expressions, closures, and the common higher-order functions that are the basis for powerful declarative database query languages (Kato et al., 1990; Trinder, 1990; Wong, 2000).

Python also has other benefits, such as clean syntax, simplicity, object semantics, ease of use and installation, community support, and extensibility (van Rossum and Drake, 2001). Encapsulation is also an important mechanism for large-scale systems. As make shows, derivations can be encapsulated in multiple ways. First, the graph of dependencies and details of construction can be hidden in rules. Second, intermediate state obtained in the process of these derivations can also be hidden. This approach is naturally extended by object oriented programming, which again is available in environments like *Python.*

Implementing knowledge maps with graph warehouses

Bioinformatics in-the-large requires complex data management for which existing relational database systems have proven to be inadequate. The apparent mismatch between the needs of bioinformatics and capabilities of existing systems is a fundamental obstacle to progress in the field.

There are many database architectures that have been proposed for bioinformatics. Specific among these are *graph databases.* Having an in-the-large perspective is very useful, since it underscores that an extension of existing graph databases is needed for scalability. We sketch one possible extension of graph databases, called *graph warehouses*, in which scaling concerns can be addressed directly while remaining consistent with the notion of knowledge flow.

Knowledge maps as graph databases. The idea of using graph databases for bioinformatics (Graves et al., 1995, 1996) is appealing, given their intuitive notation, flexibility, consistency with the navigation demands of biological databases, and ability to rigorously model weakly-structured information. It appears that graphs can support bioinformatics in ways that a classic relational database cannot.

Research on graph database systems flourished in the early- and mid-1990s, much of it emphasizing visual interfaces and pattern-oriented query languages. Significant systems included GOOD (Gemis et al., 1993), GraphDB (Gueting, 1994), GRAS (Kiesel et al., 1995), Hy+ (Consens et al., 1994), and Hyperlog (Levene and Loizou, 1995).

Graph databases like these permit implementation of knowledge maps like the *GeneMine* catalog. In addition, for complex knowledge maps they can offer benefits over the graph programming frameworks just mentioned—GVF (Marshall et al., 2003), GTL (GTL website), etc.—at the cost of sacrificing some expressiveness and flexibility.

Graph databases from an in-the-large perspective. A natural question is whether graph databases can provide a complete information management solution for large-scale bioinformatics. An in-the-large perspective is very useful in identifying limitations of existing graph databases as a solution, and in underscoring issues that a solution must address:

- For bioinformatics, there are often major differences between the *conceptual representations* of information (conceptual graphs) and the *physical representations* of information (actual graph data structures). For example, biological sequence information can be voluminous, and unoptimized representations can prevent efficient operation. Abstraction mechanisms support such differences.
- There seems an inherent tension between the strength of graphs for capturing all kinds of information, and the weakness of graphs for hiding information. The lack of abstraction of flat graph models, such as GOOD (Gemis et al., 1993), renders them of limited value in-the-large. Ideally, a graph database should admit *hierarchical graph structures* where nodes themselves contain graphs. Some encapsulation mechanisms appears to be essential for bioinformatics in-the-large.

- Graph database systems introduce limits to extensibility. The universe of tools and online resources is expanding rapidly and straightforward (programmable) access to the graphs is essential. Unfortunately, these tools and resources are often better modeled as "flows" than as graphs. In addition, writing graph algorithms is error-prone, and many graph algorithms are difficult to optimize.
- Designs for graph query language quickly face strong conflicts between expressiveness and efficiency. A general graph query database model is likely to have terrible performance. Many graph search problems scale badly and have poor worst-case performance. In fact, even visual query interfaces scale badly: finding good layouts for displaying graphs becomes computationally intractable when the graphs become large (Marshall et al., 2003).

Graph warehouses. In order to support data mining and large-scale information management for bioinformatics, as well as manage the knowledge map, we can combine the approaches above with a data warehouse architecture to give a *graph warehouse.* Again, an in-the-large perspective emphasizing scalability is helpful, and it suggests the following principles:

- **Represent graphs as flows.** For large-scale computations, it is advantageous to treat graphs as *containers*, using the container or collection classes available in modern languages such as *C++*, *Java*, or *Python.* This representation has been used successfully in large-scale graph programming frameworks (Marshall et al., 2003; GTL web site; GFC web site; LEDA web site). The container API is comfortably close to for-loops, designed to provide sequential access to a stream of objects. In this way, containers implement flows.

  Containers also provide a convenient abstraction for indexing. Index structures are powerful and compact representations for graphs, and this is important for bioinformatics. For example, sequences are often implemented with suffix trees, and can also be implemented as ordered collections of intervals. Furthermore, sequences can then be conceptualized as graphs, but actually accessed using index structures—such as indexes that provide efficient implementation of interval predicates. How to best represent a graph as an index is not always clear, multiple representations may be needed, and the representation may need to change over time to better suit evolving queries. Nevertheless using indexes in this way is convenient, and again naturally fits the model of flow.

  As flows, containers can easily implement intensions (views), rather than just extensions (collections of data). This idea has been developed for containers in the VTL and Views extensions of STL (see URLs at the end). With genomic sequences, for example, intensions can provide abstraction and encapsulation, as well as performance-improving capabilities ranging from data compression to proxy-like middleware wrapping.

  Finally, treating graphs as flows (containers, indexes) provides a solid foundation for query mechanisms. The "iterator" (loop index, or cursor) used to enumerate the elements of a container can takes on a new role for graph containers as a *traverser.* Furthermore, the traverser can serve as a form of control structure (program logic) for a mapping or algorithm to be applied to a graph—the graph analogue of the well known higher-order functional *map.*

- **Exploit knowledge maps for optimization.** Knowledge maps can be used to eliminate search in queries where the type of the desired result is known—and graph navigation can reduce to a sequence of index (graph container) lookups. Even when the type of the result is unknown, the knowledge map can be used to identify relevant types, more complex search problems can be simplified in a comparable manner.

  The knowledge map can also record other properties regarding dependencies or graph structure that are useful in processing queries. Graph structure in genomics data is often limited, ranging from trivial structures (sequences) to somewhat more complex (DAGs). These properties can be directly exploited in large-scale graph warehouse applications. The data are sufficiently complex to require a general representation but simple enough to be handled efficiently.

- **Support pipelines and parallelism.** Given the storage efficiencies mentioned above and a 'read mostly' user behavior, many graph query pipelines can be processed entirely in memory. Commodity motherboards that support multiple gigabytes of memory are inexpensive and widely available. For example,

the estimated three billion bases in the human genome can be encoded in less than a gigabyte of memory. Other genomes of interest are ten- to a thousand-fold smaller. Although there will always be problems that go beyond what can be done in memory, many "large" graph problems do not. Furthermore, graph searches are often embarrassingly parallel. It is no secret that bioinformatics is amenable to cluster computing, and although graph algorithms are not usually thought of as parallelizable, genomic graph queries can have abundant parallelism.

CONCLUSION

We have argued the significance of a fundamental shift in bioinformatics, from in-the-small to in-the-large. Adopting a large-scale perspective is a way to manage the complexity of tools and data formats that burdens many bioinformatics researchers today, automating away concerns that distract from doing science.

Where bioinformatics in-the-small is about data and tools, bioinformatics in-the-large is about metadata and dependencies. Dependencies represent the complexities of large-scale integration, including the requirements and assumptions governing the composition of tools. The *make* utility illustrates the effectiveness of managing dependencies in this way, and it encourages viewing bioinformatics in-the-large as a process of managing *knowledge maps*: meta-databases that describe types, flows, and dependencies of interest.

Many of the lessons in this paper are instances of well-known large-scale systems design principles for *scalability*:

- *Scalable design*: Separate metadata from data (defining a knowledge map) in a way that provides abstraction, encapsulation, and definition of dependencies.
- *Scalable operation*: Use metadata to provide automated compilation and optimization of queries (and automated derivation of dependencies), scalable high-performance execution using independent and pipeline parallelism, and enforcement of dependencies.

These principles have been at the very heart of large-scale software engineering models, such as the UML (see URL at the end), and configuration management systems, as well as modern database systems. In the same way, the principles are at the heart of the knowledge maps developed here, and generally characterize an "in-the-large perspective."

An in-the-large perspective has been useful to us in large-scale bioinformatics projects. Two case studies illustrated this: *GeneMine*, an interactive data-mining tool for biologists to analyze gene and protein structure-function (Lee and Irizarry, 2001), and our single nucleotide polymorphism (SNP) discovery system (Irizarry et al., 2000), which has produced about a quarter of the total coding region SNPs. Both efforts benefited from explicitly defining and managing knowledge maps, and following large-scale systems design principles. These principles also give insights about proposed architectures for information management in bioinformatics such as *graph databases*, and about the *graph warehouses* described here.

If the lessons of software engineering and database systems engineering are any guide, the shift from in-the-small to in-the-large will be of major importance in the evolution of bioinformatics. An in-the-large perspective will be a key advantage in the next phase of bioinformatics development.

ACKNOWLEDGMENTS

Work was supported by NSF grant IIS 0082964. Project home page: www.bioinformatics.ucla.edu/leelab/db.

REFERENCES

DEREMER, F., and KRON, H.H. (1976). Programming in the large versus programming in the small." IEEE Transactions on Software Engineering **2,** 80–86.

LEE, C and IRIZARRY, K. (2001). The GeneMine system for genome/proteome annotation and collaborative datamining. IBM Systems J **40,** 592–603.

IRIZARRY, K., et al. (2000). Genome-wide analysis of single-nucleotide polymorphisms in human expressed sequences. Nature Genetics **26,** 233–236.
MODREK, B., et al. (2001). Genome-wide detection of alternative splicing expressed sequences of human genes. Nucleic Acids Research **29,** 2850–2859.
FELDMAN, S.I. (1979). Make—a program for maintaining computer programs. Software—Practice & Experience **9,** 1979.
ORAM, A., and TALBOTT, S. (1991). *Managing Projects with* Make, 2nd ed. (O'Reilly Press, Sebastopol, CA).
BENDIX, L. Configuration management and version control revisited [Dissertation]. (1994). Universität-GH-Siegen, Germany.
GUNTER, C.A. (2000). Abstracting dependencies between software configuration items. ACM Transactions on Software Engineering and Methodology **9,** 94–131.
LUMPE, M., ACHERMANN, F., and NIERSTRASZ, O. (2000). A formal language for composition. In: *Foundations of Component Based Systems.* G. Leavens and M. Sitaraman (eds). (Cambridge University Press, Cambridge), 69–90.
KNIGHT, S. (2001). SCons design version 0.05. Available: scons.sourceforge.net.
MILLER, P. (2002). Cook: a file construction tool, reference manual, version 2.19. Available: www.canb.auug.org.au/~millerp/cook/cook.html.
KATO, K., MASUDA, T., AND KIYOKI, Y. 1990. A comprehension-based database language and its distributed execution. Presented at the 10th International Conference on Distributed Computing Systems.
TRINDER, P. (1990). Comprehensions—a query notation for DBPLs. Presented at the 1990 Glasgow Database Workshop.
WONG, L. (2000). *Kleisli,* a functional query system. Journal of Functional Programming **10,** 19–56.
VAN ROSSUM, G., and DRAKE, Jr. F.L. (2001). *Python* tutorial, version 2.2. Available: www.python.org.
VAN ROSSUM, G., and DRAKE, Jr., F.L. (2001). *Python*/C API reference manual, version 2.2. Available: www.python.org.
YEE, K., and VAN ROSSUM, G. (2001). Iterators." *Python* enhancement proposal 234. Available: python.sourceforge.net/peps/pep-0234.html.
SCHEMENAUER, N., PETERS, T., and HETLAND, M. (2001). Simple generators." *Python* enhancement proposal 255. Available: python.sourceforge.net/peps/pep-0255.html.
GRAVES, M., BERGEMAN, E., and LAWRENCE, C. (1995). Graph database systems. IEEE Engineering in Medicine and Biology Magazine **14,** 737–745.
GRAVES, M., BERGEMAN, E., and LAWRENCE, C. (1996). A graph conceptual model for developing human genome center databases. Computers in Biology & Medicine **26,** 183–197.
GUETING, R. (1994). GraphDB: a data model and query language for graphs in databases. Available: citeseer.nj.nec.com/156025.html.
GEMIS, M., et al. (1993). GOOD: a graph-oriented object database system. ACM SIGMOD Record **22,** 505–510.
LEVENE, M., and LOIZOU, G. (1995). A graph-based data model and its ramifications. IEEE Trans. Knowledge & Data Engineering **7,** 809–823.
MARSHALL, M.S., et al. (2003). An object-oriented design for graph visualization. Available: gvf.sourceforge.net/GVF.pdf.
KIESEL, N., et al. (1995). GRAS, a graph-oriented (software) engineering database system. *Information Systems* **20,** 21–52.
CONSENS, M., et al. (1994). Architecture and applications of the Hy+ visualization system. IBM Systems J. **33,** 458–476.
CVS. www.cvshome.org.
Subversion. subversion.tigris.org.
GTL (Graph Template Library). infosun.fmi.uni-passau.de/GTL.
GFC (Graph Foundation Classes for Java). www.alphaworks.ibm.com/tech/gfc/.
LEDA (Library of Efficient Data types and Algorithms). www.mpi-sb.mpg.de/LEDA/.
VTL (View Template Library). www.zib.de/weiser/vtl/.
Views (a C++ STL extension). www.zeta.org.au/~jon/STL/views/doc/views.html.
UML (Unified Modeling Language) Resource Center. www.rational.com/uml.

Address reprint requests to:
Dr. D. Stott Parker
Computer Science Department
University of California
Los Angeles
Los Angeles, CA 90095-1596

E-mail: stott@cs.ucla.edu