The VLDB Journal (2005) 14(3): 318–329
DOI 10.1007/s00778-005-0153-9

REGULAR PAPER

John S. Conery · Julian M. Catchen · Michael Lynch

Rule-based workflow management for bioinformatics

Received: 14 October 2004 / Accepted: 1 July 2005 / Published online: 9 September 2005
© Springer-Verlag 2005

Abstract We describe a data-centric software architecture for bioinformatics workflows and a rule-based workflow enactment system that uses declarative specifications of data dependences between steps to automatically order the execution of those steps. A data-centric view allows researchers to develop abstract descriptions of workflow products and provides mechanisms for describing workflow steps as objects. The rule-based approach supports an iterative design methodology for creating new workflows, where steps can be developed in small, incremental updates, and the object orientation allows workflow steps developed for one project to be reused in other projects.

Keywords Workflow · Rule-based system · Bioinformatics

1 Introduction

A typical bioinformatics project involves the use of several different applications, including programs that search sequence databases (e.g. BLAST [1]), align pairs of sequences (e.g. CLUSTALW [2, 3]), create phylogenetic trees from sets of aligned sequences (e.g. MrBayes [4]), and, in many cases, new applications designed for a specific project. For small projects, these applications can be run "by hand." A scientist can run a program, transfer results into a spreadsheet or text file, and cut outputs of one program and paste them as inputs to another. For larger projects, such as genome-scale analyses involving thousands of sequences and dozens of applications, researchers need to automate some or all of these steps through the use of a *workflow management system* that launches applications and passes results of one application on to others.

J. S. Conery (✉) · J. M. Catchen
Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, USA
E-mail: conery@cs.uoregon.edu

M. Lynch
Department of Biology, Indiana University, Bloomington, IN 47405-3700, USA

The workflow for a large project is unlikely to be specified and implemented in a single, straightforward process. In software engineering terms, a traditional waterfall development process, involving successive phases of requirements analysis, specification, implementation, and testing, will almost certainly fail. Instead the project will more likely follow an *iterative design process*: researchers will implement the first step in the workflow, perhaps running the application several different times in order to explore combinations of input data and parameters, and after each run, examining the output to make sure the application works as expected. Then another step will be developed, perhaps by writing a script that launches the first application, extracts results, and passes them to the second application. The creation of each step involves considerable effort in making sure the application being added to the workflow is producing results that are useful. Not surprisingly, this might cause earlier steps to be revisited, for example to change operating parameters (e.g. choose a different sequence similarity cutoff for BLAST outputs), or to include different output (e.g. to save the BLAST alignments along with other data), or maybe even to change the application altogether (e.g. use a different BLAST implementation [5] or different similarity search method [6]).

In this paper we introduce a method for organizing bioinformatics workflows that is intended to support this sort of incremental development. In our approach, the focus is on work products as opposed to process control. We introduce a software architecture called the *data-centric pipeline* that provides a project framework in which the products of each step of the workflow are stored in a database, and the workflow is managed by a rule-based system that uses declarative specifications of data dependences between steps to automatically order the execution of the steps and the corresponding database updates.

One of the ways a rule-based approach supports the development of new workflows is by providing support for *reusability*. Rules are inherently modular, and a rule that is developed for one project can be fairly easily incorporated into another project. A rule-based workflow management system provides another dimension of reusability by exploiting a common pattern for a workflow step. This pattern can be encapsulated and used as a default command in the body of a rule. Researchers can define new steps by writing class descriptions that build on the basic pattern, reusing the methods that apply to their project and overriding or extending others.

In the first part of this paper we survey related work on methods for managing workflow, focussing on systems intended specifically for bioinformatics. Following that is a brief description of a small bioinformatics project that will be used as a running example throughout the paper. As an introduction to relational databases for biologists who may not be familiar with database technology, and to justify our building the rule-based workflow management system on a database foundation, Sect. 4 describes the advantages of a relational database system for bioinformatics projects. Section 5 describes the rule-based workflow model and the software architecture it is based on, and in Sect. 6 we briefly describe the pipeline interface program (PIP), a prototype implementation of the model that has been used successfully in several of our bioinformatics projects. Section 7 describes the base class that encapsulates the common pattern that occurs in bodies of rules and how one can derive new objects that can override any or all of the base class methods. The paper concludes with a short discussion and plans for future work.

2 Related work

Software systems that manage complex workflows have proliferated in the last 10–15 years. Many systems, including several commercial applications, have been developed specifically for business process management [7]. To help establish standards in the field, and to help users develop processes that can be controlled by variety of workflow systems, the Workflow Management Coalition (WFMC) was formed to promote interoperability, and in 1994 published a workflow reference model [8].

Workflow management is increasingly important in scientific research. Early systems (e.g. LabBase [9]) were developed mainly to coordinate applications within local networks or supercomputer centers, and many used a database to store work products [10]. One such system, developed by Ailamaki et al. [11], used the database schema to represent a project's workflow. With the advent of computational grids [12], several groups began developing workflow management systems for distributed applications (e.g. GridDB [13], WF-Pilot [14], POESIA [15], and Taverna [16]).

Computational biology is an increasingly important application area for workflow management [17], especially since data sources and applications are widely dispersed across the internet. Several new systems have been developed for bioinformatics workflows in the last 2 years alone, including HyperThesis [18], BioPipe [19], BioWBI [20], BioMake [21], Taverna [16], and Pegasys [22].

Almost all of these systems use a straightforward and intuitive "dataflow" model for describing workflows: a process is represented as a node in a directed graph, and an arc $A \rightarrow B$ connects processes A and B if the output of A is used as an input to B. The software system that controls the workflow (an *enactor* in the terminology of the WFMC reference model) uses the graph to schedule processes and pass data between them. Advanced workflow systems provide several useful features, including the ability to interact with users in partially automated workflows that require human intervention, to run independent jobs in parallel, and to checkpoint and restart after hardware failures.

Many of the systems for managing bioinformatics workflows help researchers develop new workflows through the use of a graphical interface. In Taverna, for example, there is a palette with icons representing commonly used applications [16]. The user can place instances of a process in the workflow by selecting an icon from the palette, and processes can be connected by clicking on instances in the workflow document. In systems with GUI interfaces workflow descriptions can be saved, usually as XML files, so they can be recalled and edited at a later time. Many other enactors also use XML as the formal representation of workflows.

Not all enactors require workflows to be defined by acyclic graphs. In an iterative workflow, output from a node may feed back to a predecessor. This implies the use of conditional execution and the ability to direct outputs along different paths, e.g. when a node has several output arcs the process may send its results along all paths or choose among different paths.

Workflows defined by graphs can become very complex, especially when the graphs are cyclic, making it very difficult to analyze the workflow, e.g. to predict if or when a process will be activated. One way to manage this complexity is to constrain the graphs in some form. van der Aalst et al. defined a set of *workflow patterns* and evaluated a number of workflow systems in their ability to express and manage these patterns [23]. The Biology Workflow Builder (BioWBI [20]) limits workflow components to five simple patterns, and uses an algebra to define rules for composing these patterns into a larger workflow. Another approach is to use a different formal model as an alternative to dataflow graphs. For example, the YAWL system [24] is based on Petri nets, which provides a framework for analyzing various properties of workflows.

The rule-based workflow management scheme presented in this paper provides another alternative formalism. In a rule-based workflow, individual steps are described by rules. As in other rule-based systems, rules can have dependences, i.e. a rule header can name other steps that must complete before the rule can be executed, and a workflow enactor will use dependences between rules to schedule activities. The main advantage of a rule-based approach over either structured or unstructured flow graphs or Petri nets is that rules describing workflow products provide a *data-centric* view of the workflow. By focussing on the data objects produced by each step, a workflow specification can be a concise declarative description of the experimental protocol [13]. BioMake [21] is another rule-based enactor, where statements written in a control language based on the Prolog programming language [25] define steps and manage the workflow. Davulcu et al. [26], Bonner [27], and Senkul et al. [28] defined formal logic systems to analyze properties of workflows describe by graphs, and these systems all have a similar structure to rule-based systems, but we are not aware of any workflow enactors based on these logics.

For bioinformatics, an important benefit of a rule-based framework is the support it provides for iterative design of workflows. As an example, consider how BLAST [1] might be incorporated into a workflow. An iterative development process would first consider how to obtain the inputs, in this case FASTA-formatted sequences, so the user would first write a query to fetch sequences from the database, and if necessary rewrite them in FASTA format. The next step is to figure out how to launch BLAST and pass it the sequences. The first few runs may generate full "BLAST reports" as text output so the user can verify BLAST is working as expected, but eventually the workflow developer will determine a format for storing BLAST hits in the database. BLAST can be run with a parameter that has it generate tab-separated records, or maybe a special wrapper will be written that parses the output to extract the information needed for this workflow. Finally, the researcher may want to experiment with different runtime parameters, for example different similarity cutoffs, substitution matrices, gap penalties, etc. It is also quite likely the user may come back to the design of the BLAST step at some future time, after other steps that consume BLAST outputs have been implemented. For example, it may be necessary to alter one of the BLAST parameters or to change the code that processes BLAST outputs to capture part of the output that was not saved previously.

The important point is that a workflow enactor can help this design process if it allows a researcher to execute individual steps or restart the workflow at arbitrary points. In process-centric workflows selecting individual steps or restarting the workflow can be difficult because most systems lack an explicit representation of the state of the workflow [7]. For a data-centric, rule-based workflow, however, restarts are straightforward: the user identifies the work product that should be rebuilt, and the system infers the processes that need to execute in order to regenerate the data.

Two other ways the rule-based approach supports workflow development are through modularity and reusability. Rules provide a complete and concise specification of workflow products, including the inputs, outputs, and commands used to implement each workflow step, and it is a fairly simple operation to copy a set of rules from one workflow to another. In Sect. 7 we describe how rule-based workflow supports reuse through inheritance, where a base class captures a common rule structure, and researchers can write new classes that extend or modify this structure for the specific needs of new workflows.

3 Example project: searching for tandem duplicates

To introduce the rule-based approach to workflow management we will first describe a simple bioinformatics project. The project involves only a few steps, but these steps illustrate the sorts of operations and data flow that are commonly found in bioinformatics.

The project is a search for pairs of genes called *tandem duplicates*. When chromosomes are passed from one generation to the next, the cells carrying the chromosomes have one copy of each gene from the parent cell. The chromosomes in daughter cells are often the result of recombination, when information from the parent's chromosomes is reorganized so the daughter cell carries genes from two different parent chromosomes (Fig. 1). Normally, the chromosomes break and recombine at the same location. But if the breaks happen at different locations one of the daughter cells may end up with two copies of a parent's gene—a tandem duplication.

A research group that wants to study instances of tandem duplicate genes in yeast (*Saccharomyces cerevisiae*) could begin the project by searching for pairs of genes that might be tandem duplicates using the following workflow:

1. Build a table with a list of yeast chromosomes and the web addresses of FTP servers from which to download genome files. This table will have 16 records, one for each yeast chromosome.
2. Download the annotated genome files, which will be text files containing descriptions of the chromosomes, including all the genes. This step will download 16 files, ranging in size from 500 KB to 3.5 MB.
3. Scan the chromosome files to extract the complete set of gene sequences from each chromosome. This step will put around 6,000 gene sequences in the database.
4. Create a BLAST database from the genes, and then do an "all *vs.* all" BLAST search. This step runs 6,000 BLAST searches, comparing each gene against all the others. Command line options can be used to tell BLAST to print only the two best hits for each input and to ignore lower quality matches that are unlikely to be duplicate

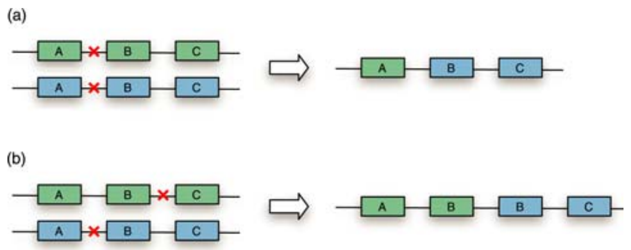


Fig. 1 Tandem duplication. Genetic recombination in a diploid organism (an organism with two copies of each chromosome). **a** Normal crossover: Two chromosomes break at the same relative location, so the crossover produces the same number of genes on the new chromosome. **b** Unequal crossover: The new chromosome is the result of splicing together the two longer portions of each original chromosome, and has two copies of gene *B*

genes. The result of each search will be a pair of IDs of genes that are most similar to the input gene. Depending on the settings of BLAST parameters, this step will generate a table containing around 9,000 BLAST hits (each gene will match itself, and about half the genes will have a high quality match to another gene).
5. Look for "reciprocal best hits" in the BLAST results, that is, genes X and Y such that $X \neq Y$, Y is the best hit for X and X is the best hit for Y.
6. Filter the set of reciprocal best hits to find pairs that are on the same chromosome and within a specified distance of each other.

The types of applications in this example project are fairly typical: they include a commonly used application (BLAST) that can be downloaded from NCBI [29] and installed on the user's own system, some special purpose scripts (e.g. building the table of chromosome names and web addresses), and scripts that use library routines (e.g. one can use the `Bio::Perl` [30] library to parse the chromosome descriptions to get the gene sequences). A more realistic search for tandem duplicates would involve many other steps, perhaps using applications that do a complete alignment of the candidate pairs (e.g. CLUSTALW [2, 3]) and a more rigorous analysis of the similarity of two sequences (e.g. PAML [31]), but the steps listed earlier are varied enough to illustrate the main features of rule-based workflow management.

The description of the steps in the tandem duplicate workflow assume the programs will run on the user's own system, or perhaps a local server within their organization. Most users will also store the data objects produced by the applications in a local database, but since most database systems use a client–server model, the database server could be anywhere. Regardless of the location of the database server, the user retains local control over the data.

An alternative to running applications locally and to storing all work products in a locally controlled database is to use distributed resources found elsewhere on the internet. A web services approach might use web service composition techniques (e.g. [32]) to organize distributed resources into a project workflow. For example, a researcher could use a web services composition language to define a workflow so that the step that runs BLAST passes sequences from a gene database at a genome center to a BLAST service at a third site. While there are many advantages to using distributed resources – such as being able to tap into the most recent genomic data and not worrying about keeping a local copy up to date, or being able to run complex applications on high performance computers instead of being forced to install them locally – there are several reasons to adopt a workflow system that manages all the data at a single location:

– Storing a local copy of each work product helps researchers validate the workflow, e.g. by doing an "audit" after the workflow has completed to check the accuracy of intermediate steps, or to track the progress of a particular data item. In the tandem duplicate example, a researcher might know before the project begins that a particular gene is or is not one of a tandem pair, and one way to check the accuracy of the workflow is to track the processing of these genes through the entire workflow.
– Local control of data provides better support for iterative development, giving users the flexibility to run applications several different times (as in the description in the introduction of how BLAST might be added to a workflow).
– Most projects will need to do some amount of special-purpose analysis, such as filtering or transforming data produced in one step so it can be used in later steps. The reciprocal best hits step of the tandem duplicate project is an example. This "computational glue," which will most likely be special-purpose scripts or programs written specifically for the project, will need access to the data produced at various steps.
– Many projects will be working with data that is not available on the internet, but is produced locally. A good example might be a laboratory that is sequencing the genome of an organism for the first time, so the data is not available at any of the genome centers. After developing and testing the tandem duplication pipeline on yeast data, the group will be ready to search for tandem duplicates in the new genome by running it on their own new genomic data.
– Bioinformatics projects often deal with very large data sets. It may be impractical to wrap this data in XML to transfer it between web services, or the project may require more processing than a web service may be willing to provide. The yeast tandem duplication project, as simple as it is, requires around 6,000 BLAST searches.

An orientation toward local control over project data does not preclude a hybrid approach in which some of the data used in a project is fetched from a remote site (the yeast tandem duplicate project begins with downloading data from a genome center) or some steps involve running applications on remote sites. If a rule-based workflow does use distributed resources, however, the data produced by a step is brought into the project database where it will be available for analysis and use in later steps.

4 A case for relational databases in bioinformatics workflows

Since the software architecture of our rule-based workflow model requires the use of a database to store work products, it is worth exploring some of the benefits of using a relational database management system in a bioinformatics project. This section, which can be skipped by readers who are already familiar with databases, is a brief overview of relational database management systems, in particular the open source MySQL [33] system, and the advantages of using MySQL or similar systems in bioinformatics projects.

Many scientists, when they first learn how a database system like MySQL is organized, are not convinced the apparent complexity is worth the effort. Instead of being immediately available for viewing and editing in a text file, data is now hidden away on a database server, and the only access is through a special new interface that will force them to learn a new language.

For all but the smallest projects, however, a database system has several benefits:

– The database provides a uniform interface to the data, and thus an organizing structure for applications that will be used in the workflow. For small projects, where researchers run familiar applications like BLAST, CLUSTALW, or PAML by hand, there is no problem leaving the data in the text files generated by those applications – relevant items can be cut and paste into spreadsheets or other analysis programs. For larger projects, where there are many more applications or where the applications will be run several times, researchers will be writing scripts to run the applications and parse the results. This is where the database becomes useful: Instead of deciding on a file format for the output that will be saved for each application, and developing code to read and write those formats so the data can be passed from one application to the next, the scripts can adopt a common structure that reads text fetched from the database (usually tab separated the records) and generates output in that same form.
– The structure of the tables that store the data can be changed without affecting applications that use the data. For example, suppose the application that extracts gene sequences from the genome files in the tandem duplicates project has to be modified because a new application is being added to the workflow, and this application needs gene attributes that were not used in the first version of the workflow. The researcher can revise this step, and add one or more columns to the table that stores gene records, without worrying about other applications that read the gene records. The BLAST step might not have to modified at all, since all it needs from the genes table is an ID and a sequence, and the query to fetch that information will not be affected by adding a new column to the genes table. But if gene descriptions had been stored in a text file, the code in the script that reads gene descriptions to make the BLAST database will almost certainly have to be updated to parse the new file format.
– If the workflow is defined so that one step is to process only part of the data created earlier, the logic that filters the data for the step can be implemented in the database query instead of the script that implements the step. For example, according to the specification given earlier, pairs of genes in the reciprocal best hits table of the tandem duplicates workflow will show up twice, once as (X, Y) and once as (Y, X). To analyze each pair only once, use only records where $X < Y$. When the data is in a database, it is very easy to add this constraint to the query that fetches data to pass to the application that will use it; when the data is in a text file, any application that uses the data would have to do its own filtering. Adding the constraint to the query is more efficient, simpler to implement, and easier to maintain than code inside applications that read the data.
– Database queries can also be very useful for initial analysis of the output of a step, to make sure the application implementing the step is working as expected. In previous projects on the evolution of duplicate genes [34, 35] we originally used K_a and K_s, metrics of sequence similarity [36], to estimate the age of the duplication events and the divergence of the gene pairs. Later we decided to use a different application, which computed two different measures, known as d_N and d_S [31]. In situations like this it is very easy to write queries in SQL that look for outliers generated by each method (e.g. "how many pairs have $d_S > 5.0$?") and to see if the results between the methods are in general agreement (e.g. "in how many pairs do the two methods produce very different results, where $|K_s - d_S| > 1.0$?"). These simple analyses are extremely useful when implementing a workflow step, to make sure an application is giving expected results on test cases, or maybe to look for interesting cases to subject to further analysis in later steps. For more complex situations MySQL can export data to spreadsheets or statistics packages such as R [37] for analysis or visualization.
– Although there is some overhead in learning to use SQL, it can be quite helpful, and in many cases fairly complex operations can be implemented directly in SQL so that it is not necessary to write separate applications for some steps in the workflow. For example, the reciprocal best hits step in the tandem duplicates project can be implemented completely by SQL queries.

Finally, the fact that database systems such as MySQL use a client–server architecture is also very useful, particularly when projects involve teams from different organizations. Instead of e-mailing result files back and forth, project members can all access the same data directly from the database server. CGI scripting languages, such as PHP [38], have facilities for accessing data in relational databases, so team members can also view results via their web browsers.

5 Software architecture: the data-centric pipeline

Our approach to building a workflow management system for bioinformatics is based on a *software architecture* that supports the gradual development and refinement of project workflows. The term 'software architecture' is used in a variety of contexts in computer science; here we use it to mean a framework that provides "constraints on the form and structure of a family of architectural instances" [39]. The architecture defines the general structure of a system, for example a client–server architecture for a distributed system, or a model-view-controller architecture for an application with a graphical user interface, and then particular systems are constructed using the architecture as a model.

Our software architecture, the *data-centric pipeline*, is characterized by the use of a database to store workflow products and a rule-based workflow enactment scheme that automatically schedules workflow steps based on declared dependences between workflow products:

– The workflow will use a database to hold inputs to applications, and to store the results generated by applications; when the database is a relational database each step of the workflow will generate a database table.
– There is a one-to-one relationship between tables and steps: Each step of the workflow produces a complete table, and no other step will modify the table.
– Applications are organized so they read a stream of objects fetched from a database and generate a stream of objects that are stored back in the database. With respect to a relational database, this means a query, possibly joining information from several tables, generates a stream of records for the application, and the application produces a stream of records that will be stored in a new table.

The requirement that every workflow step produce a new table may seem restrictive, but in practice we have found it to be very effective at simplifying the development and maintenance of workflows. The overall philosophy is that each workflow step generates new information, and the best way to manage the data for the project is to add the information generated by each step to a monotonically increasing collection of records. The high-level specification of some workflow steps may seem at odds with this requirement. For example, suppose a workflow needs to filter the set of genes extracted from a genome file, e.g. to filter out pseudogenes or other sequences so they are not used in later steps. One approach is to implement a step that deletes these records from the gene table, but in the data-centric pipeline architecture the workflow would use a step that generates a new table (the "suspects" table) containing the IDs of sequences that should not be used. Later steps that use gene records would see records from a virtual table, the result of a query that fetches all genes from the gene table that are not also in the suspects table. The query to fetch genes to use is more complicated – in MySQL it might involve a left join of the original table and the IDs in the suspects table – but in the long run the workflow is easier to maintain. As another example of a virtual view, in the previous section we described a project that first used a quantity known as K_s as a metric of sequence similarity, but then later switched to using a different measure known as d_S. Instead of overwriting old K_s values with newer d_S values, one would just have the new workflow step write its own table of d_N and d_S values. Steps that have sequence similarity as an input would not be affected by the change; all that is necessary is to modify the query that selects records passed as inputs to the applications, for example with a join that combines gene IDs with d_S from the new table.

There are three main reasons for preferring to keep all the data from each step, and to implement deletions and updates through virtual views. First, keeping all the data makes it easier to "roll back" and redo earlier steps. For example, the criteria that define suspect genes may change, so that genes not used earlier are now included. If all the genes are still in the original table, one only has to change the query that fetches them, or the application that creates the suspects table, but the entire gene table would have to be rebuilt if the suspects have been deleted. Second, having access to all the data means it is possible to compute a wider range of statistics, e.g. to compute both the average gene length for all genes in the genome and for all genes used in the project (i.e. all non-suspect genes). Third, it is possible to implement the sorts of "audits" and comparisons mentioned in the earlier sections – one can compare K_s with d_S for all pairs, or track the progress of a single gene, even it is in the suspect table, as it progresses through the workflow.

There is also a practical reason for requiring the database to keep a monotonically increasing data set, where each workflow step creates a new table: it means a workflow enactor can use timestamps on tables to automatically schedule the execution of the steps. The workflow enactor described in the next section uses a simple rule structure to define the dependences between data stored in tables. When a table is updated, the enactor can determine which other steps must then be re-executed to bring the project up to date. If a table is modified by more than one step, for example if it is created in one step and a later step deletes a subset of the records, the data dependences are much more complicated.

Not all of the project data has to actually be stored in the database. Some steps may create or download files that are kept in the project directory instead of the database, but in these cases the workflow step should put file descriptions in the database. There is an example of this design in the tandem duplicate workflow, where the download step retrieves files from a remote server and records file descriptions in the database.

The third aspect of the data-centric pipeline – that applications read and write streams of tab-separated records – also entails some initial extra work. In many cases, it will be necessary to write "wrappers" to run widely used applications, such as CLUSTALW and PAML. A wrapper will read a stream of input records, use data elements to define input parameters for the application, run the application, and then generate the stream of output records from the outputs of the application. Although this appears at first to be a serious drawback, the class library from the Open Bioinformatics Foundation [30] makes it very easy to write these wrappers. The library has routines for running a wide variety of common bioinformatics applications.

6 The pipeline interface program

To evaluate the data-centric pipeline architecture and the concept of a rule-based workflow we developed a simple system called PIP (for Pipeline Interface Program). PIP is written in Perl and interfaces to a MySQL database.

6.1 Rule syntax

PIP rules are very much like rules in a Unix makefile. A rule has a header, consisting of the rule name and a list of dependences on the first line, followed by body that consists of zero or more lines containing commands that will be executed when the rule is invoked. The collection of rule headers is the declarative specification of the work products of the workflow: the name of the rule corresponds to the name of a table in the database, and the dependence list is the set of names of other tables that contain the data required to construct the table. When the user asks PIP to create a new table, or to update an existing table to a new version, PIP will construct a work schedule based on the rule headers, and only steps for tables that directly or indirectly contribute information to the requested table will be executed.

As an example, here is the outline of the final rule from the tandem duplicates project, the rule that selects a subset of the records in the reciprocal best hits table to make the final table of results:

```
tandems:    rbh genes
    mysql yeast -e 'DROP TABLE IF EXISTS tandems'
    mysql yeast -e 'CREATE TABLE tandems SELECT ...'
```

The header shows this is a rule to build a table named `tandems`, and that it depends on information in the `rbh` and `genes` tables. The two lines in the body of the rule are both shell commands, in this case commands which pass queries to MySQL. The first query deletes an old version of the table if there is one, and the second query builds the `tandems` table. The body of the second query (not shown) joins information from the two input tables to build the new table.

The commands in the body of a rule can be Unix shell commands, or, as described later in Sect. 7, an invocation of a Stage object. Any program that can be run from a Unix shell can be used in a rule body. The only requirement for the commands in the body of a rule are that after the last command has terminated there needs to be a new table in the database with the same name as the rule; in the previous example, note there is a `CREATE TABLE` command to build the table named `tandems`.

The commands in the body of a rule are not limited to MySQL operations, and commands may have "side effects" that read or write files in the project directory. An example from the tandem duplicates project is the rule that runs BLAST – what NCBI calls the BLAST database is a set of binary files, and one of the commands in the body of the rule for the BLAST step uses an NCBI program named `formatdb` to create these files for BLAST.

Some applications already generate outputs as tab-separated records that can be loaded directly into a MySQL databases. Others, such as programs in the PAML package, will need simple wrappers to run the application and reformat the results. This is a straightforward operation for many applications; for example, a Perl programmer would include the `Bio::Perl` library [30] and create a factory object to launch the application.

When a rule is invoked, the system compares the timestamp on the table to be built with the timestamps on the dependences. If any one of the dependences has a newer timestamp, it presumably has updated information, and the commands in the body of the rule are executed. A key point in using rules to manage workflow is that when the rule for a table is invoked, the system does a recursive invocation of the rules for each table listed in the dependence list. Thus, if a table containing information generated early in the workflow is updated, all the user needs to do is invoke the rule for the final product, and the workflow is automatically restarted.

Dependence processing is best explained in terms of a directed acyclic graph, or DAG. There is one node for each rule, and for any rule r with dependences $d_1, d_2, \ldots$ there is an edge connecting each of the d_i to r. When a rule is invoked, the system traverses the graph, working back through the dependences for that rule. The DAG for the tandem duplicates workflow is very simple, since the dependences mostly just define a linear list where step i depends only on step $i-1$. An example of a richer set of dependences from a project that builds a database of tRNA genes is shown in Fig. 2.

A node may be encountered more than once during the traversal, but as long as the graph is acyclic the dependence checking terminates and each workflow step is executed just once. In the example shown in Fig. 2, suppose a new set of genome sequences becomes available, and the `source` table is updated. Next suppose the researcher wants information from the `codons` table, so PIP is invoked to make sure that table is up to date. PIP will check `DNA`, and see that it depends on `download`. Since `download` depends on `source`, and `source` has recently changed, PIP runs the applications in the `download` step, and then, since `download` is now newer than `DNA`, it runs the applications in the `DNA` step. Since the `codons` table also depends on `genes`, the dependence checking has to work up that branch of the DAG also. But now when PIP gets to the `download` table, it sees it is up to date (the timestamp on `download` is newer than the timestamp on `source`), so PIP only runs commands in the body of the `genes` step. Now that `DNA` and `genes` have both been updated PIP can run

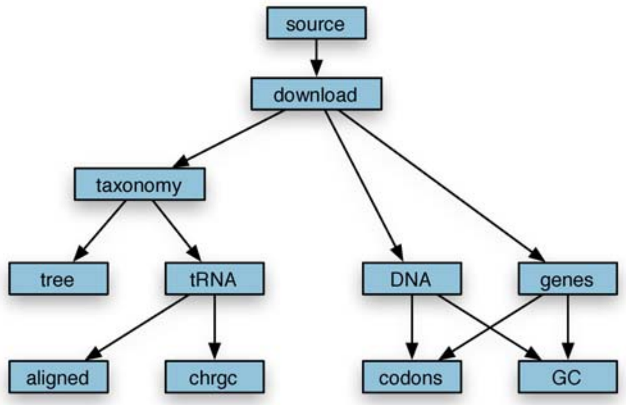


Fig. 2 Dependence graph for a tRNA database

the applications in the body of the `codons` step. As a final note, the `taxonomy` table and the tables below it, on the left side of the figure, are now outdated, but they will not be updated until there is a request for information that depends on one of these tables.

6.2 Control file

The rules that describe steps in the workflow are collected into a control file called a "pipfile." When the workflow is activated, PIP reads the pipfile and builds a DAG from the rules. The pipfile is also a place where the researcher can define workflow parameters, such as the name of the database to use, database connection parameters (e.g. username and password, if they are to override global settings), and directories where applications can be found.

When PIP is invoked from the command line the user can specify the name of the rule that should be executed. For example, to invoke the rule in the example above, one would type

```
% pip codons
```

6.3 Using PIP to develop workflows

To add a new step to a workflow, the developer has four basic tasks:

- write a query that selects the information from existing tables that will be passed to the application executed by the new step;
- test the invocation of the application, if necessary writing a wrapper that will use the information supplied by the query;
- define the table that will contain the output of the application, and make sure the output records from the application or its wrapper conform to the table definition;
- collect the first three parts and incorporate them into a rule, and insert the rule into the pipfile.

Two facilities built into PIP are very useful when implementing and testing the new step.

The first is the ability to force the execution of a rule, whether or not the corresponding table is outdated with respect to its dependences. When the `-r` (restart) option is specified, PIP will process only the rule named on the command line, and will execute all the commands in the body of the rule, even if the table for the rule is newer than any dependence.

The second feature is the ability to test the pipeline without actually executing any commands or updating any tables. When called with the `-n` (norun) option, PIP will construct a DAG for all the rules, as usual, and begin processing the rule named on the command line. It will check dependences, but instead of executing commands in the bodies of rules, PIP just prints the commands in the terminal window.

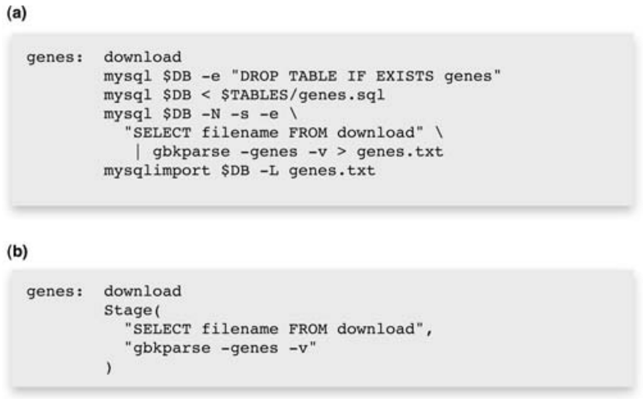
A common scenario for adding a new step to a workflow managed by PIP is thus:

- develop the new step using the four tasks outlined earlier;
- use PIP to execute the step using the `-r` option as necessary;
- use MySQL queries to examine the results, e.g. looking for outliers, or selecting results from known test cases, or generating data to be passed to a visualization program;
- if implementation of the new step leads to a modification of an earlier step, make the required changes, and invoke the modified step with the `-r` option;
- check the effects this modification has on the remainder of the pipeline by typing `pip -n`;
- analyze the transcript to see how the change to the earlier stage propagates to other stages, and adjust those as necessary;
- run PIP on the full workflow to bring final output tables up to date.

7 Stage objects and inheritance

Most steps in a rule-based workflow follow a consistent pattern: an output table is prepared, either by creating a new table or erasing the contents of an existing table; a query is executed, and the results of the query are piped to a set of shell commands, which implement the workflow step; the output of the command is filtered and reformatted and added to the database. We call this pattern the "prep, step, and grep" pattern ("grep" being the name of a common Unix utility that filters text streams).

To allow workflows to take advantage of this pattern, researchers can use techniques from object-oriented programming to instantiate an object called a Stage. The object constructor needs the two pieces of information that distinguish one step from another: the MySQL query that fetches records for the step, and the commands that launch the applications used in the step.



(a)

```
genes:  download
        mysql $DB -e "DROP TABLE IF EXISTS genes"
        mysql $DB < $TABLES/genes.sql
        mysql $DB -N -s -e \
          "SELECT filename FROM download" \
            | gbkparse -genes -v > genes.txt
        mysqlimport $DB -L genes.txt
```

(b)

```
genes:  download
        Stage(
          "SELECT filename FROM download",
          "gbkparse -genes -v"
        )
```

Fig. 3 Using a stage object to define a workflow step. **a** The original step uses four shell commands to initialize a table, run a script named `gbkparse`, and put the output into the database. **b** When using an object constructor the workflow developer just specifies the query and the shell command. The database name and directory with MySQL table definitions must be defined in the control file

```
.require Blast.pm

blast:  genes
        Blast(
          "SELECT orf, sequence FROM genes \
             WHERE length(sequence) > 20",
          "mysql2fasta | blastall -p blastp \
             -d $BLASTDB -b 2 -m 8 -e 1e-40",
          "$BLASTDB"
        )
```

Fig. 4 Example of a derived class

The PIP prototype allows users to instantiate and invoke Stage objects. To use an object in a workflow, the body of the rule consists of a single statement that invokes the constructor (Fig. 3). When PIP sees an object constructor in the body of a rule, it replaces it with an automatically generated set of commands. These commands create a new instance of the Stage object, passing it the name of the database and the location of a directory that has project table definitions, both of which are defined in the header of the pipfile. When the rule is activated, the command calls a method named `exec`, which in turn calls, in order, methods named `prep`, `step`, and `grep`. Any of the four methods defined in the base class can be extended or replaced in new object derived from the Stage class.

Figure 4 shows an example of the invocation of a new class written for the tandem duplicate project. The `blast` step of the workflow is almost like all the other steps, but it has the extra requirement of building a BLAST database before running BLAST itself. The file named `Blast.pm` contains the definition of the new `Blast` Stage object, and this object has a new implementation of the `prep` method that calls `formatdb`. The rest of the step is the same as any other step implemented by a Stage object, so `Blast.pm` does not have definitions for any other methods, i.e. it inherits their behavior from the base class.

8 Discussion

This paper describes a new paradigm for organizing bioinformatics workflows based on a software architecture called a "data-centric pipeline." The workflow is specified by a set of rules, where each rule describes a table in a database, and a workflow enactor uses dependences between rules to schedule steps. Applications launched by the enactor read streams of records generated by a query and write streams of records to be stored in a table. We also described a simple prototype enactor named PIP, which was developed to test this new framework. PIP was strongly influenced by the Unix make utility. When PIP is run, the user specifies the name of a table to update, and PIP uses rule dependences to automatically schedule the operations necessary to update that table.

PIP has been used successfully in several of our own projects, including a datamart for bacterial tRNA genes (the workflow shown in Fig. 2), assembly and annotation of a newly sequenced fungus genome, and a project that is building a database of human genes and pairs of their co-orthologs in zebrafish. As an example of how simple the command language is, the control file for a project that is comparing orthologous genes from closely related bacterial species has 33 rules and contains 296 lines of text (half of which are comments). The first step in the workflow generates a table of genomes available at NCBI [29]. The next few steps download chromosome descriptions and parse the files to get lists of genes and other features. The system then automatically creates BLAST databases, runs BLAST, aligns results with CLUSTALW, computes d_N and d_S values with PAML, and runs various other applications written specifically for this project. We wrote 21 wrappers to launch the applications that do not already read and write tab-separated text files. The longest and most complex of these has 320 lines of Perl code (including comments), and the average is 95 lines.

PIP adds a very small overhead to projects. It is itself little more than a wrapper that invokes the applications according to command lines found in the project control file. The only thing that makes PIP less efficient than a special purpose script written to manage the workflow is the cost of reading and parsing the rules in a control file, building a DAG from rule dependences, accessing the project database to get the timestamps on the tables that correspond to the nodes in the DAG, and then traversing the DAG. One way to measure this time is to invoke PIP on a project that has just been completely updated. PIP will build and traverse the complete DAG, and then print a message saying all steps are up to date; the time to execute this "null" workflow is then the sum of the execution times of each overhead component.

As shown in Fig. 5, the percent of time spent processing the rules is negligible compared to the total time to complete the workflow. Running the complete tandem duplicate workflow on all 16 yeast chromosomes requires a little over 30 min (the row labeled "full" in the figure). The PIP overhead (the line labeled "null") is less than a third of a second.

Another question to ask about performance is whether the data-centric pipeline, which uses a relational database server, is less efficient than a special purpose script that leaves all the information generated by applications in flat files. A rough estimate of the overhead of querying the database for input data for a step, plus the overhead of

Operation	Wall Clock	CPU
full	30:13.00	26:23.00
null	00:00.32	00:00.24
`download`	01:03.76	00:03.84
`genes`	00:42.34	00:38.41
`blast`	27:32.35	27:14.93
`rbh`	00:00.55	00:00.27
`tandems`	00:12.49	00:00.28

Fig. 5 PIP performance on the tandem duplicates workflow

creating a table and writing results to the table, can be obtained by running each workflow step individually. The difference between the wall clock and CPU time would mainly be the result of the CPU that executes the workflow waiting for the database server to respond. The last five rows of Fig. 5 show there is some overhead. Overheads in a special-purpose script would be harder to quantify, but would have to include time for file I/O plus time to parse the files – one advantage of the database approach is that the database server does most of the parsing and other file management.

The command language for PIP is intentionally very simple. It does not allow more than one step to run at any one time, either in parallel or overlapped (e.g. a pipeline where streams of outputs of one step are passed to the next step, with both running concurrently), and it does not have built-in facilities for checkpointing or other support for large, distributed projects. PIP has primitive support for case variables, which are values generated by the workflow and used by subsequent steps [8], but they are not used to implement conditional execution of rules or other control constructs.

Parallelism, iteration, conditional execution, and other more complex control can be implemented in a rule-based framework, but this would require the use of case variables and a more full-featured workflow control language. For example, rules could take the form of guarded Horn clauses [40], where a system evaluates guards (boolean conditions) at the front of each rule, and then use values of case variables to select one of the rules whose guard clause is satisfied as the rule to execute. An alternative approach, which we plan to investigate, is to compile the very simple rules of PIP into an XML specification so they can be executed by Taverna, BioPipe, Pegasys, or other systems that do provide checkpointing, rollback, and other more advanced workflow management.

While PIP itself does not try to exploit parallelism in the processing of rules, the commands it runs can be parallel applications. PIP can invoke any application, whether it is sequential or parallel, as long as it has a command line interface. The only limitation is that they have a Unix command line interface. An example of a rule body that uses multiple processors is the CLUSTALW step in the bacteria ortholog project mentioned earlier. We wrote a new stage class named `Parallel` that overrides the base class methods for launching the application and collecting results. The new class splits the input stream into independent substreams and launches a separate instance of the CLUSTALW wrapper on each substream. By using secure shell (`ssh`) or other applications for running programs on remote hosts a PIP workflow can use Global Grid resources, perhaps fetching data from a remote site or running an application on another host.

The ability to encapsulate workflow steps into abstract descriptions is a goal that is shared by several projects working on semantic web services for bioinformatics [32, 41, 42, 43]. An example of how a rule-based workflow provides an abstract view of a step can be seen in the BLAST example mentioned in Sect. 1: a project may need to do some sort of sequence similarity search, and store the results in a database. If the step that does the search is specified in the form of an abstract rule, one just needs to modify the rule body to use a different application, e.g. WU-BLAST [5] or FASTA [6]. From the perspective of the workflow enactor, all that is required is that some application provide a similarity search to fill the specified table.

In our PIP implementation of the rule-based approach, where rule bodies contain Unix commands, it would be straightforward to take advantage of semantic web services; all that is needed is a rule body that would invoke a sequence similarity server on a remote host. Lord et al. make the case for a programmatic interface to web services [44], and any service that can be invoked from a command line can be incorporated into a PIP workflow.

Perhaps the biggest difference between workflows managed by PIP and workflows inferred by web services composition techniques [32, 42, 43] is that the latter are often oriented toward single transactions. A common example is a travel support service, where a user connects to a virtual service that has been composed from several independent web services, e.g. for airline reservations, hotels, and sightseeing. Workflow in this context refers to how a request from a single user is processed by passing objects between the constituent services. In this situation, the performance overheads of handling transactions are a concern, since prompt feedback to the user is paramount. In many cases, a bioinformatics workflow could also be transaction oriented, particularly if a researcher is working on a small data set. But many workflows are similar to the one used as an example in this paper. In these workflows, large amounts of data are completely processed at each step before the next step is invoked. In these situations, overhead from managing individual steps is not as critical.

One of the main motivations for the rule-based approach was to support an iterative design process for bioinformatic workflows, where modularity, reusability, and the ability to isolate and test individual steps by executing them repeatedly are expected to be especially beneficial. This sort of iterative design strategy for workflows is likely to be helpful in other fields as well. Although it was designed for bioinformatics projects using MySQL databases, a rule-based enactor could be used in any field where work products can be stored in a database (relational or object oriented), where the database provides mechanisms for checking the status of items in the database, such as timestamps on tables, and where applications can be run via scripts that read and write items from the database.

Availability

PIP is available for download from `http://teleost.cs.uoregon.edu/PIP`. The distribution contains the source code and documentation, including a tutorial (the tandem duplicates project) that illustrates many of the features of PIP.

Acknowledgements We thank the anonymous reviewers, who made several constructive suggestions, and members of the Wolfe and McLysaght labs at Trinity College Dublin, "early adopters" who have provided valuable feedback. Research projects that use PIP, and the development of PIP itself, were supported by grants from the National Science Foundation (MCB-0342431, JSC and ML) and the National Institutes of Health (2R01-RR10715, JSC and JMC). JSC also thanks Science Foundation Ireland, which provided support through an E. T. S. Walton Visitors Award for the 2004–2005 academic year.

References

1. Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. Nucleic Acids Res. **25**(17), 3389–402 (1997)
2. Chenna, R., Sugawara, H., Koike, T., Lopez, R., Gibson, T.J., Higgins, D.G., Thompson, J.D.: Multiple sequence alignment with the CLUSTAL series of programs. Nucleic Acids Res. **31**(13), 3497–3500 (2003)
3. Thompson, J.D., Higgins, D.G., Gibson, T.J.: CLUSTALW: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Res. **22**(22), 4673–4680 (1994)
4. Huelsenbeck, J.P., Ronquist, F.: MrBayes: Bayesian inference of phylogenetic trees. Bioinformatics **17**(8), 754–775 (2001)
5. Lopez, R., Silventoinen, V., Robinson, S., Kibria, A., Gish, W.: WU-Blast2 server at the European bioinformatics institute. Nucleic Acids Res. **31**(13), 3795–3798 (2003)
6. Pearson, W.R., Lipman, D.J.: Improved tools for biological sequence comparison. Proc. Natl. Acad. Sci. U.S.A. **85**, 2444–2448 (1988)
7. van der Aalst, W., van Hee, K.: Workflow Management: Models, Methods, and Systems. MIT Press, Cambridge, MA (2002)
8. WFMC: Workflow reference model. Technical report, Workflow Management Coalition, Brussels (1994) http://www.wfmc.org/standards/model.htm
9. Goodman, N., Rozen, S., Stein, L.D., Smith, A.G.: The LabBase system for data management in large scale biology research laboratories. Bioinformatics **14**(7), 562–574 (1998)
10. Medeiros, C.B., Vossen, G., Weske, M.: WASA: A workflow-based architecture to support scientific database applications (extended abstract). In: Database and Expert Systems Applications, pp. 574–583 (1995) citeseer.ist.psu.edu/bauzermedeiros95wasa.html
11. Ailamaki, A., Ioannidis, Y.E., Livny, M.: Scientific workflow management by database management. In: Rafanelli, M., Jarke, M. (eds.) SSDBM, pp. 190–199. IEEE Computer Society (1998)
12. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure, 2nd edn. Morgan Kaufmann, San Francisco, CA (2003)
13. Liu, D.T., Franklin, M.J.: GridDB: A data-centric overlay for scientific grids. In: Proceedings of the 30th VLDB Conference, pp. 600–611 (2004)
14. Altintas, I., Bhagwanani, S., Buttler, D., Chandra, S., Cheng, Z., Coleman, M., Critchlow, T., Gupta, A., Han, W., Liu, L., Ludäscher, B., Pu, C., Moore, R., Shoshani, A., Vouk, M.A.: A modeling and execution environment for distributed scientific workflows. In: SSDBM, pp. 247–250. IEEE Computer Society (2003)
15. Fileto, R., Liu, L., Pu, C. et al.: POESIA: An ontological workflow approach for composing Web services in agriculture. VLDB J.: Very Large Data Bases **12**(4), 352–367 (2003)
16. Oinn, T., Addis, M., Ferris, J., Marvin, D., Greenwood, M., Carver, T., Pocock, M.R., Wipat, A., Li, P.: Taverna: A tool for the composition and enactment of bioinformatics workflows. Bioinformatics (2004)
17. Jagadish, H.V., Olken, F.: Data management for the biosciences: Report of the NSF/NLM workshop on data management for molecular and cell biology. Technical report, LBNL-52767. Lawrence Berkeley National Laboratory (2003)
18. Bhowmick, S.S., Vedagiri, V., Laud, L.: HyperThesis: the gRNA spell on the curse of bioinformatics applications integration. In: Proceedings of the 2003 ACM International Conference on Information and Knowledge Management (CIKM-03), pp. 402–409. ACM Press, New York (2003)
19. Hoon, S., Ratnapu, K.K., Chia, J.M., Kumarasamy, B., Juguang, X., Clamp, M., Stabenau, A., Potter, S., Clarke, L., Stupka, E.: BioPipe: A flexible framework for protocol-based bioinformatics analysis. Genome Res. **13**(8), 1904–1915 (2003)
20. IBM: Bioinformatics workflow builder (BioWBI) (2004). http://www.alphaworks.ibm.com/tech/biowbi
21. Mungall, C.J.: BioMake: functional logical task management for bioinformatics. In: Bioinformatics Open Source Conference (BOSC'04), Glasgow, Scotland (2004) http://open-bio.org/bosc2004/accepted_abstracts.html
22. Shah, S.P., He, D.Y. et al.: Pegasys: Software for executing and integrating analyses of biological sequences. BMC Bioinformatics **5**(4) (2004)
23. van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Dist. Parallel Databases **14**(1), 5–51 (2003)
24. van der Aalst, W.M.P., Aldred, L., Dumas, M., ter Hofstede, A.H.M.: Design and implementation of the YAWL system. In: Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAISE'04), pp. 142–159. Springer-Verlag, Heidelberg (2004)
25. Sterling, L., Shapiro, E.: The Art of Prolog: Advanced Programming Techniques. The MIT Press, New York (1986)
26. Davulcu, H., Kifer, M., Ramakrishnan, C.R., Ramakrishnan, I.V.: Logic based modeling and analysis of workflows. In: ACM Symposium on Principles of Database Systems, pp. 25–33 (1998)
27. Bonner, A.: Workflow, transactions, and datalog. In: ACM Symposium on Principles of Database Systems, pp. 294–305 (1999)
28. Senkul, P., Kifer, M., Toroslu, I.H.: A logical framework for scheduling workflows under resource allocation constraints. In: Bernstein, P.A. et al. (eds.) Proceedings of the Twenty-Eighth International Conference on Very Large Data Bases (VLDB'02), pp. 694–705. Morgan Kaufmann, Los Altos, CA 94022, USA (2002)
29. National Center for Biotechnology Information. http://www.ncbi.nih.gov
30. Open Bioinformatics Foundation. http://www.open-bio.org
31. Yang, Z.: PAML: a program package for phylogenetic analysis by maximum likelihood. Comput. Appl. Biosci. **13**(5), 555–556 (1997)
32. Yang, J., Papazoglou, M.P.: Service components for managing the life-cycle of service compositions. Inform. Syst. **29**(2), 97–125 (2004) http://dx.doi.org/10.1016/S0306-4379(03)00051-6
33. MySQL. http://www.mysql.com
34. Lynch, M., Conery, J.S.: The evolutionary fate and consequences of duplicate genes. Science **290**(5494), 1151–1155 (2000)
35. Lynch, M., Conery, J.S.: The evolutionary demography of duplicate genes. J. Struct. Funct. Genomics **3**(1–4), 35–44 (2003)
36. Li, W.H., Wu, C.I., Luo, C.C.: A new method for estimating synonymous and nonsynonymous rates of nucleotide substitution considering the relative likelihood of nucleotide and codon changes. Mol. Biol. Evol. **2**(2), 150–174 (1985)
37. The R Project for Statistical Computing. http://www.r-project.org
38. PHP: Hypertext Preprocessor. http://www.php.net
39. Garlan, D., Perry, D.E.: Introduction to the special issue on software architecture. IEEE Trans. Software Eng. **21**(4), 269–274 (1995)
40. Ueda, K.: Guarded horn clauses. In: Proceedings of the 4th Conference on Logic Programming, pp. 168–179. Springer Verlag, New York (1986)

41. Hashmi, N., Lee, S., Cummings, M.P.: Abstracting workflows: unifying bioinformatics task conceptualization and specification through semantic web services. In: W3C Workshop on Semantic Web for Life Sciences. Cambridge, MA (2004)
42. Hull, R., Su, J.: Tools for design of composite web services. In: Weikum, G., König, A.C., Deßloch S. (eds.) SIGMOD Conference, pp. 958–961. ACM (2004)
43. Orriëns, B., Yang, J., Papazoglou, M.P.: Model driven service composition. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC. Lecture Notes in Computer Science, vol. 2910, pp. 75–90. Springer-Verlag, Berlin Heidelberg New York (2003)
44. Lord, P. et al.: Applying semantic web services to bioinformatics: Experiences gained, lessons learnt. In: ISWC'04. LNCS 3298, pp. 350–364. Springer-Verlag, Berlin Heidelberg New York (2004)