

The LabFlow System for Workflow Management in Large Scale Biology Research Laboratories

Nathan Goodman¹, Steve Rozen², Lincoln D. Stein³

<http://jura.wi.mit.edu/rozen>

(1) The Jackson Laboratory, Bar Harbor ME, nar@jax.org

(2) Center for Genome Research, Whitehead Institute, Cambridge MA, steve@genome.wi.mit.edu

(3) Cold Spring Harbor Laboratory, Cold Spring Harbor NY, lstein@cshl.org

Abstract

LabFlow is a workflow management system designed for large scale biology research laboratories. It provides a workflow model in which objects flow from task to task under programmatic control. The model supports parallelism, meaning that an object can flow down several paths simultaneously, and sub-workflows which can be invoked subroutine-style from a task. The system allocates tasks to Unix processes to achieve requisite levels of multi-processing. The system uses the LabBase data management system to store workflow-state and laboratory results. LabFlow provides a Perl5 object-oriented framework for defining workflows, and an engine for executing these. The software is freely available.

Introduction

Workflow management is an attractive technology for constructing laboratory information management systems (LIMS) for large scale biological research projects. Using workflow management, a developer can construct a LIMS whose structure mirrors the laboratory protocol in a natural way. We can depict such systems as diagrams (cf. Figure 1), reminiscent of flowcharts or finite state machines, in which the nodes represent stages of the protocol, and the arrows indicate the order in which stages occurs.

Large scale biological projects typically involve a combination of laboratory procedures and computational analyses. Laboratory procedures often involve a mixture of manual and automated methods, while computational analyses use a combination of internally and externally developed software, and public or proprietary Web servers. These tasks are connected to form a "protocol" indicating the or-

der in which the steps must be performed. These protocols often have multiple branch points and cycles. Some tasks, esp. computational ones, are simple and fast, while others may last for hours or days. Laboratory procedures inevitably fail from time-to-time, and software occasionally crashes; graceful handling of such failures is essential. Compounding the problem is a rapid rate of change: research laboratories continuously strive to refine old techniques and invent new ones in the never ending struggle to do the best possible science.

This article describes a laboratory workflow management system we have developed called LabFlow. LabFlow provides an object-oriented framework (Fayad and Schmidt 1997) for describing workflows, an engine for executing these, and a variety of tools for monitoring and controlling the executions. LabFlow is implemented in Perl5 (Stein 1996; Wall, Christiansen and Schwartz 1996) and is designed to be used conveniently by Perl programs. LabFlow runs on top of a laboratory data management system that we have developed, called LabBase (Goodman et al. 1998; Goodman, Rozen and Stein 1998), which in turn runs on Sybase (McGovern and Date 1992); with modest effort, LabFlow could be ported to other data management systems. The software is freely available.

The software described in this article is the latest in a series of workflow management systems we have built over a period of almost 10 years (Goodman, Rozen and Stein 1994; Stein et al. 1994; Stein, Rozen and Goodman 1994; Rozen, Stein and Goodman 1995). We and our colleagues have used the predecessor systems in genetic and physical mapping of mouse (Dietrich et al. 1996; WICGR 1997), genetic mapping of rat (Steen 1997), and physical mapping of human (Hudson et al. 1995). The current system is now

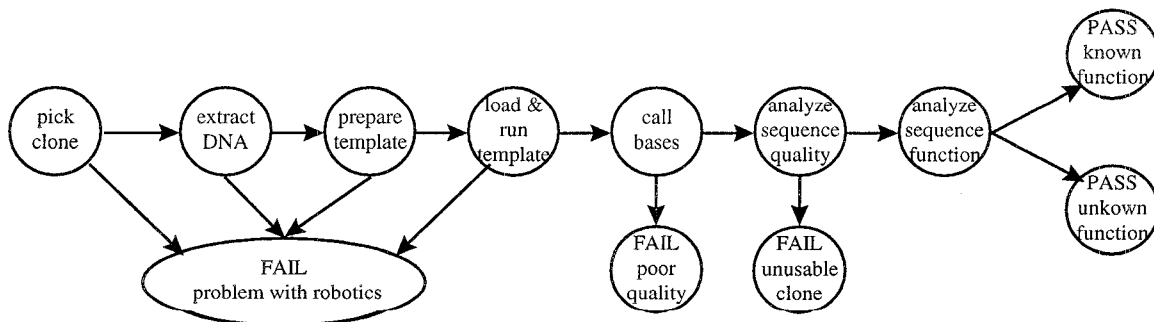


Figure 1: Sample Workflow. Single Pass EST Sequencing.

in the final throes of development and will enter service soon.

When we began our efforts, workflow management was not a recognized topic of study. Our first efforts were inspired by Shlaer and colleague's insightful book on object life cycles (Shlaer, Mellor and Mellor 1992). Since then, the field has evolved considerably and is now both an active commercial sector (cf. (WfMC 1996; Anaxagoras 1997; WARIA 1998)) and research area (cf. (Hsu 1995; Mohan 1998)). Remarkably, although LIMS is also an active commercial sector (cf. (LIMSource 1998)), there is almost no overlap among these areas; to our knowledge, there is no commercial LIMS product that supports workflow management in any general sense, although we know of one company, Cimarron Software (Cimarron 1998), that develops custom-LIMS employing this technology.

This article describes the design and implementation of LabFlow. Section 2 defines the LabFlow workflow model. Section 3 explains how we map the workflow model onto process and data services available to us. Section 4 describes the object-oriented framework that realizes these ideas. LabFlow is research software and is incomplete in many ways. As we go, we will endeavor to point out the major areas that need further work.

Workflow Model

A workflow depicts a process, i.e., an activity that occurs over time, and a *workflow model* is a formalism for specifying workflows. The study of processes is a central theme in computer science, and many formalisms exist for specifying processes. Our workflow model is built upon well-known process models from computer science but is specialized for the problem at hand. As with most process models, our model includes *syntax*, which describes the form of a workflow, and *semantics*, which defines how a workflow executes.

We first describe a baseline model to introduce the main ideas. We then discuss two extensions (parallelism and sub-workflows) that are implemented in LabFlow, and two extensions (multiple input workflows and grouping of samples) that are not.

Baseline Model

Syntactically, a *workflow* is a directed graph. The nodes, called *Steps*, represent discrete tasks in a laboratory protocol, and the arcs, called *Routers*, control the order in which Steps can execute. In essence, Steps embody the work and Routers the flow of the workflow. An *Initial Step* is one with no incoming arcs, and a *Terminal Step* is one with no

outgoing arcs. Figure 1 above illustrates a workflow for a simple EST sequencing project generating and analyzing single pass sequences from a large number of cDNA clones. The Steps in the workflow are: (i) pick clone, which is the Initial Step, (ii) extract DNA, (iii) prepare template, (iv) load & run template, (v) call bases, (vi) analyze sequence quality, (vii) analyze sequence function, and (viii-xii) Terminal Steps for various success and failure conditions.

Semantically, workflow execution is controlled by objects, called *tokens*, that flow through the workflow. Intuitively, a token represents one sample being tracked by the LIMS. A token may be *present* at one Step at a time (this is relaxed in the next section). When a token is present at a Step, the Step may *fire* and do its task using the token as a parameter. When the Step finishes, it stores its results in the token, and passes it to the Routers emanating from the Step. Each Router examines the token and either passes it to its successor Step or does nothing. If several Routers are able to pass the token, only one does so (but see next section). If no Router forwards the token, it remains in the Step, but does not cause repeated firing. Initial Steps create tokens and Terminal Steps consume them: each Initial Step may create any number of tokens, all at once or over time; when a token reaches a Terminal Step it stays there forever since a Terminal Step has no successors. We expect that many tokens will be present in a workflow simultaneously at the same or different Steps. In all cases, tokens execute independently; the presence of additional tokens has no effect on the correct processing of any individual one.

The baseline model handles the example of Figure 1 quite well. As clones become available, the Initial Step, "pick clone", creates a token for each one. Assuming the Step is successful, the outgoing Router passes the token to the next node, "extract DNA". When DNA extraction completes successfully, the outgoing Router pass the token (which now represents the DNA extracted from the clone) to "prepare template". Execution proceeds in this manner until the token reaches one of the "PASS" or "FAIL" Terminal Steps.

Parallelism

We now extend the example slightly to obtain sequence-reads from both ends of each clone. This leads to the diagram in Figure 2 in which the protocol splits after DNA-extraction into parallel paths for each clone-end, and re-joins after analysis of sequence quality. We have added a new Step to assemble the reads into a consensus sequence, and this consensus is now the input to the functional analysis Step.

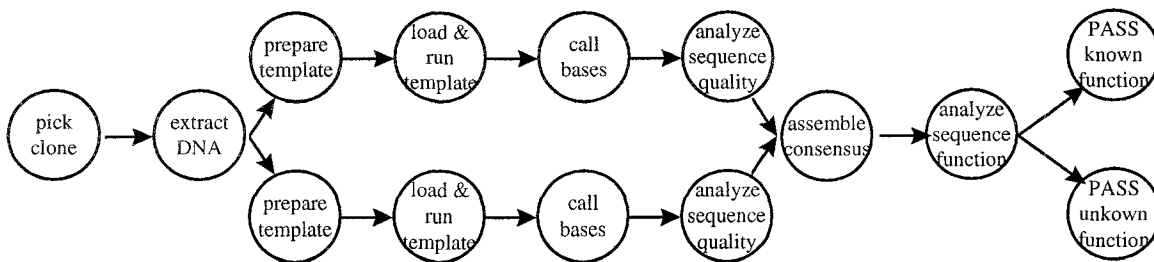


Figure 2: Two-End EST Sequencing. Failure steps are omitted for legibility.

Parallelism complicates the semantics considerably. Although it is intuitively obvious what we would like Figure 2 to mean, getting this formally correct is hard. A very basic issue is to define exactly how parallel paths re-join. In the example, it is clear that we want the “assemble consensus” Step to wait until sequence-reads arrive along both paths. But what if one path fails? If the failure is read-specific (e.g., the sequencing reaction did not work), we probably want to proceed with the one successful read, but if the failure is clone-specific (e.g., sequence analysis showed the clone to be contaminated), we probably want to abandon both paths. The semantics are further complicated if we allow the paths to split again, and especially if we introduce a loop allowing the paths to split an arbitrary number of times. For example, after base calling, if the sequence-quality is marginal, we might send the read forward to get a first pass analysis but also loop back to get a second (hopefully better) read. It is far from obvious how joining should be defined in such cases. These issues, though well-studied in the field of concurrent programming, are largely ignored in workflow management. (Two exceptions are TRiGS (Kappel et al. 1995) and OpenPM (Davis, Du and Shan 1995), although neither offers a complete solution; some discussion also appears in (Cichocki et al. 1998).)

Another complication concerns the visibility of data along parallel paths. When we split an execution, can Steps along one path see the results produced along the other? If the answer is “yes”, then the full gamut of well-known concurrency anomalies are possible (for example, the “analyze quality” Step along one path might read the base calls produced along the other path!). If “no”, we must elaborate the join operator to merge results as well as executions.

The solution in LabFlow has two parts. First, we extend the basic model so that if multiple Routers are able to pass a token forward, they all do so. This means that a token can be present at multiple Steps simultaneously. These Steps may execute concurrently or serially in any order. If a token arrives at a Step where it is already present, the system is allowed to combine the two instances into one; this is mostly an issue for Terminal Steps. Data is visible along all paths, and it is the programmer’s responsibility to avoid anomalies. Some help is provided by the underlying LabBase data management system which ensures atomicity of updates, but the main assistance comes from the second

part of the solution, namely sub-workflows, the topic of the next section.

Sub-Workflows

A *SubFlow* is a workflow that runs subroutine-style within a Step. When a token is routed to a Step containing a SubFlow, the Initial Steps of the SubFlow execute using the token as a parameter. Like Initial Steps in a regular workflow, they create one more new token that will flow through the (sub-) workflow. These new tokens are independent of, though linked to, the token in the parent workflow. As token reach the SubFlow’s Terminal Steps, a *termination procedure* associated with the SubFlow examines the execution state and decides whether all (or enough) work has been done; if so, it constructs a summary answer that merges information from all the tokens that were created on the parent token’s behalf and returns this to the parent workflow. The answer also summarizes the final outcome (e.g., PASS vs. FAIL) of the SubFlow.

In Figure 3, we show a workflow for the two-end EST example that uses SubFlows. The SubFlow, called “sequence both ends”, encapsulates the parallel processing of the two ends. For a given clone, the main workflow executes as we have seen previously until its token reaches “sequence both ends”. Then it invokes the SubFlow. The SubFlow’s Initial Step creates two new tokens, one for each clone-end, and sends each down the main sequencing path. Each token proceeds down the path until it reaches a Terminal Step, which may be “PASS” or one of several failure cases. When each token reaches its Terminal Step, the system invokes the SubFlow’s termination procedure which waits for both ends to be finished, collects the pair of results, summarizes the outcome, and passes the ensemble to the main workflow. The possible outcomes are (1) both ends were successfully sequenced, in which case the two reads proceed to “assemble consensus”; (2) one or both reads failed due to a clone-specific problem, in which case the clone is abandoned; (3) one read was successful and the other failed due to a read-specific problem, in which we keep the successful read but skip assembly; or (4) both reads failed due to read-specific problems, in which case we retry the process starting at “extract DNA”; presumably the Routers for this case would include logic to avoid retrying indefinitely.

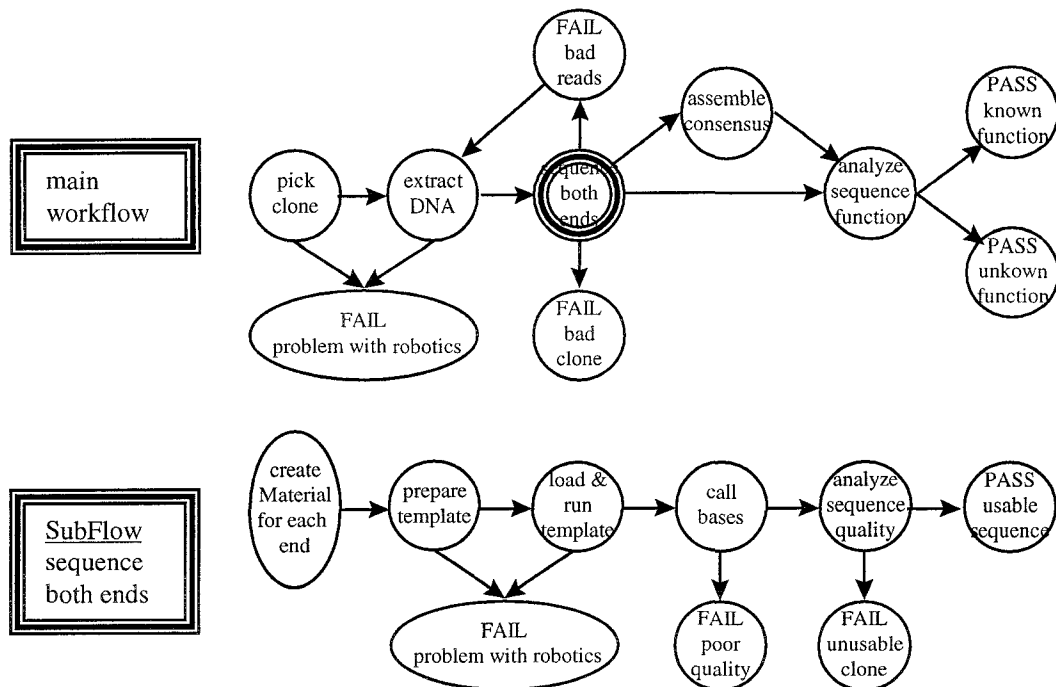


Figure 3: Two-End EST Sequencing Using SubFlows.

LabFlow also supports another kind of sub-workflow, called a CompoundFlow, in which the sub-workflow operates on the same token as the main workflow. This is useful when reusing a series of Steps in several workflows. For example, the main sequencing path from “prepare template” to “analyze sequence quality” is likely to be the same in many projects, and it makes sense to abstract it into a sub-workflow. CompoundFlows are analogous to macros (#define’s) in C and other programming languages.

Workflows with Multiple Materials

The workflow model as defined so far cannot handle protocols in which multiple samples are combined at a single Step. This is a very common case. Consider a gene-expression example in which DNA-arrays (e.g., Affymetrix or Synteni chips) are probed with different mRNA samples. In broad terms, the workflow for this process would have the following Steps: (1) prepare array; (2) prepare mRNA sample; (3) probe array with sample; and (4) analyze the results. It seems intuitive to regard the “probe” Step as having two inputs, an array and a sample. See Figure 4. To handle this, we must extend our workflow model to distinguish the two inputs, and to indicate that the Step should fire only when both inputs are present. Petri nets (Murata 1989; CPN 1998) are a well-known process modeling formalism with these capabilities; Figure 4 is drawn using the standard symbology of Petri nets. Presumably, we would also like Steps to be able to emit multiple outputs, and would need some form of data typing to ensure that outputs are connected to inputs in a consistent manner.

OPM (Chen and Markowitz 1995) and the solution from Cimarron Software (Cimarron 1998) have these capabilities.

There are many projects that start with two sets of reagents and seek to test each element of one set against each element of the other. For example, in genetic mapping, one starts with a set of markers and a set of DNA samples, and the goal is to genotype each marker against each DNA. The core of the protocol is a two-sample workflow like the one in the gene-expression example. The overall workflow must embed this core in a control structure, such as a Cartesian product or nested iteration operator, that applies the procedure to every pair of elements from the two sets. An entire procedure of this sort may be embedded in an even larger protocol that iteratively selects the sets of reagents. For example, in complex trait (QTL) analysis, one starts

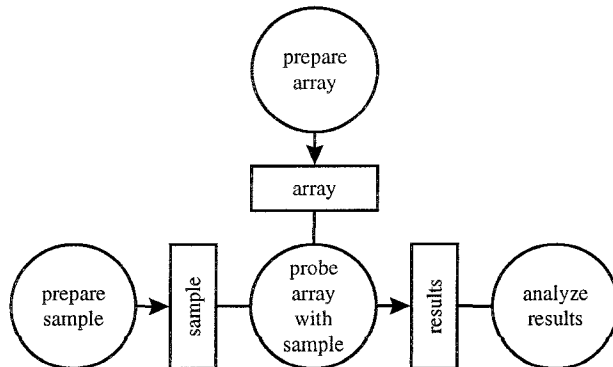


Figure 4: Gene Expression Example Using Petri Net.

with a modest number of markers and DNA samples to get a coarse mapping of the trait, and then adds additional markers and samples in a highly controlled manner to home in on the fine position. This suggests that the model needs the ability to create and manipulate sets of samples as well as individual ones.

A further nuance is whether the procedure consumes the reagents. Data is the clearest example of a non-consumed reagent. Biological samples, though finite, are often adequate for the planned course of experiments and, if so, may be considered non-consumable. In other cases, the procedure fully consumes (e.g., aliquots of chemicals) or partially consumes (e.g., a filter that may be washed and re-probed a few times) the reagents. The model should support these cases directly, lest they unduly complicate the control structure of the workflow.

Grouping of Samples

In large scale projects (and many small ones) samples are manipulated in batches. The physical form factor of these batches is an important aspect of the protocol. In our first EST-example, clones are likely to enter the protocol as colonies living in growth media, then are picked into tubes for DNA extraction, then arrayed into 96-well plates for template preparation, then loaded onto 64- or 96-lane gels for sequencing, which results in a two-dimensional image which is processed by base calling software. After all these transformations, the LIMS must be able to link each lane identified by the base calling software with the original clone. It is important that the workflow description capture this information.

In our experience, the biology of a protocol is best expressed in terms of individual samples, as we have done in our examples. But the laboratory really works on batches. Ideally, the workflow model should provide a natural way to bridge these two levels.

Process and Data Model

Having described the workflow model supported by LabFlow, the next problem is to devise a feasible strategy for implementing the model using readily available computational services.

We expect workflows to contain tens to hundreds of Steps and thousands to millions of tokens, i.e., samples. The workflow execution must persist for months or years, and recover flawlessly after crashes. Though some Steps are fast and others slow, all data must flow through the system in a timely manner. In some cases, a Step may require access to a dedicated instrument, e.g., a specific robot, and samples simply have to wait their turn; but the system should not unduly delay a token if the necessary resources are available. Distributed execution should be possible, and we would like to control where execution occurs. In some cases, it makes sense to move data to Steps, for example, if a Step requires special hardware. In other cases, it is better to leave the data in place and move

the Steps to them, for example, if the data volume is large. The system must store several kinds of information: (1) the state of the execution, i.e., where each token is in the workflow; (2) the results of the execution, i.e., the contents of each token; and (3) possibly, the state of each Step and Router if these are state-full objects.

This is a formidable list of desiderata. What we have described, in effect, is a persistent, recoverable, multithreaded distributed object system. Such systems exist in research, (e.g., Liskov's Thor (Liskov et al. 1996)), but to our knowledge no commercial products come close. Evidently, we must scale back our goals to something more achievable.

Our key simplification is that LabFlow is an object-oriented framework that supports distributed execution, but is far short of being a distributed object system.

LabFlow decomposes the Step concept into three parts. *Components* are application programs which do the actual work of the Step; these may be complex analysis programs, simple data entry screens, Web pages (really, CGIs), interfaces to laboratory instruments, or anything else. *Workers* are objects that encapsulate components; the Worker class defines interfaces that allow arbitrary programs to be connected to the workflow. *Steps* (now in the narrow LabFlow sense) are objects that glue Workers to a workflow. Components can run anywhere in a distributed system and may have persistent state, while Workers and Steps must run in the same execution context as the workflow itself and have no persistent state. LabFlow provides design support for component-distribution and state, but does not provide actual mechanisms for these features, leaving this (potentially difficult job) to application programmers.

LabFlow runs a workflow by invoking its Steps. By default, the system initiates a Unix process to run each Step; the workflow developer may override this by grouping Steps into *execution units* and telling the software how many processes to run for each unit. Each process sits in an infinite loop executing any of its Step that has work to do, and sleeping when all Steps are quiescent. LabFlow also runs a manager-process that monitors the status of the other processes, starting new ones if any should fail, and shutting down the entire entourage on demand. By defining execution units wisely, the workflow developer can group multiple short Steps into a single process to reduce operating system overhead, or allocate multiple processes to a long-running Step so that multiple Materials can execute the Step concurrently. It would be useful to extend the manager-process to automatically start new processes for Steps that are falling behind (and reap processes for Steps that have too many), but we have not done this yet.

For persistence, LabFlow relies on LabBase (Goodman et al. 1998; Goodman, Rozen and Stein 1998), a laboratory data management system that we have developed. LabBase provides a structural object data model, similar to those found in ACEDB (Durbin and Mieg 1991), OPM (Chen and Markowitz 1995), lore (McHugh et al. 1997), UnQL (Buneman et al. 1996), and many other systems, and adds to this the concepts of Step, State, and Material. Step is a

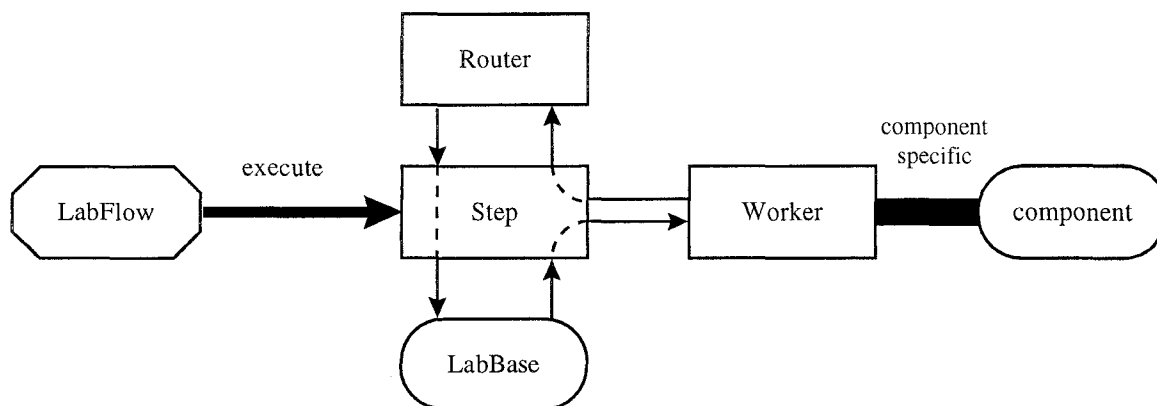


Figure 5: Collaborations Among Principal LabFlow Classes.

direct reflection of the same-named concept in LabFlow. States represent the execution-state of a workflow (i.e., which tokens are at which Steps), and Materials represent the tokens themselves. Of most interest here, LabBase operates atomically on execution-state and execution-results (using database transactions) for two purposes. One is to ensure that if multiple processes are assigned to the same Step, the processes will operate on different tokens. The second is to ensure that if the system were to crash just as a Step completes, the execution-state and results will remain consistent, allowing clean recovery. LabBase offers no explicit support for distributed databases, but since it runs on commercial relational products, it can exploit the distributed database features offered thereby.

Though far short of a distributed object system, these modest capabilities offer enough, we believe, to build usable workflows. Indeed, the software described here is more capable than previous versions that we have used successfully for this purpose.

LabFlow Framework

The most significant classes in the LabFlow framework are LabFlow and its subclasses SubFlow and CompoundFlow, Step, Router, and Worker. Figure 5 illustrates the connections among the classes and other important LabFlow elements. We will first discuss the Worker class, since the requirements for this class drive much of the design. Then, we move on to Steps and Routers, and finally LabFlow itself.

Workers

Since components are “arbitrary programs”, they lie outside the boundary of our system design. This should not blind us to the fact that components are the most important, and often most complex, elements of a LIMS, because they do the actual computational work required by the project. A LIMS without components is barren.

Workers exist to compensate for the idiosyncrasies of components. Every component needs at least one Worker, and a typical system will contain tens or hundreds of these.

The programmer who develops a Worker must be an expert user of the component in order to build effective interfaces to it, and should also have a good understanding of the project’s scientific methods. It is unrealistic to also expect the programmer to be expert in workflow and data management. Our design of the Worker class library insulates the programmer from most of these issues.

We expect developers to organize their collection of Workers as a class library. We provide a Worker base class and a few slightly specialized subclasses, and we expect developers to derive their Worker classes from these. To facilitate reuse of Workers, the system provides a layer of data translation between the objects seen by Workers and those in the database. This translation is the province of the Step class and is discussed in the next section. In the long run, success with LabFlow (and probably any other modular approach to LIMS construction) will depend on the accumulation of a well-designed library of good Workers.

The principal method that a Worker must implement is `execute`. This method takes as input a Perl5 object representing a token (or a list of such objects, see below), and returns a Perl5 object (or list of objects) as output. The Worker converts the input object to whatever format its component requires and passes it to the component. The component performs its computation and returns the results to the Worker. The Worker converts the component’s result back into a Perl5 object with the format required by our software, and returns this as its result. The LabFlow software (specifically, the Step class) takes care of retrieving the Worker’s input from and storing its output to the database.

In addition, the Worker may implement a `ready` method if special scheduling is required. By default, LabFlow will run a Worker whenever there is a token waiting for it. In some cases, additional scheduling constraints apply, for example a Worker that requires a dedicated instrument cannot run until the instrument is free. To handle this, the Worker would implement a `ready` method that checks whether the instrument is free and indicates to LabFlow whether or not the Worker can be run now. The

`ready` method can also block (i.e. suspend its execution) until the Worker is runnable.

The `ready` method can also indicate how many tokens the Worker is able to process. By default, a Worker need only process one token at a time. In some cases, a Worker may wish to process batches of tokens for efficiency or other reasons. The `ready` method can specify that the Worker only wants one token (which is the default), or it can specify the minimum and maximum batch size that the Worker can handle, or it can specify that the Worker wants to process all waiting tokens. The method can also indicate a specific list of tokens that the Worker wishes to process, in which case, the system gives the Worker the subset of the requested tokens that are still waiting for it. (Some of the tokens may already have been “grabbed” by other processes.)

Workers for Initial and Terminal Steps are a little different. Initial Workers in a top level workflow receive no input; in a SubFlow, the input is the token in the parent workflow. The output of an Initial Worker represents new tokens (which the system creates on the Worker’s behalf), rather than updates to existing ones. A Terminal Step in a top level workflow generally has no Worker; in a SubFlow, the Worker is responsible for joining parallel executions and producing a summary result for the parent workflow. By default, the Worker simply calls the termination methods associated with the SubFlow.

The Worker class defines three additional methods for handling state-full components: `start`, `stop`, and `recover`. The `start` method is invoked when the process running a Worker starts up, the `stop` method is invoked when process shuts down cleanly, and `recover` is invoked following a crash. The default implementation of these methods does nothing; the programmer developing a Worker is responsible for implementing these methods when needed.

Steps and Routers

Steps and Routers are generic. We expect application programmer to create instances of these, but not subclasses.

To create a Step instance, the programmer specifies three main properties: one is the Step’s Worker; the other two are input and output mappings that define (i) which fields of the LabBase Material representing the token should be retrieved from the database when getting the Step’s input; (ii) which fields of the Step’s output should be stored back into the database; and (iii) a translation between the field-names used by the Worker and those in the database. These field-names are generally different since Workers can be reused in multiple workflows. By default, if no mappings are provided, all fields of the Material are retrieved as input, all fields of the output object are stored into the database, and no field-name translation occurs.

To create a Router instance, the programmer specifies its source and target Steps, and provides a procedure that examines the input and output of the source Step and decides whether to route the token to the target Step. A Router may

have multiple target Steps, in which case the token is routed to all.

The Step class implements one principal method, `execute`. The `execute` method calls the Worker’s `ready` method. If the Worker says it is ready, the Step gets from the database as many waiting tokens as the Worker wants; the act of getting the tokens places each in a special “in-process” State to protect it from other processes running the same Step. The translation of input fields happens at this point. The Step then calls the Worker’s `execute` method, passing the tokens as input. When the Worker finishes, the Step takes the Worker’s outputs, translates the fields, and passes the results to its outgoing Routers.

Each Router runs its decision procedure on each result. If the procedure returns a true value, the token is routed to the Router’s target Steps. A Router can also indicate that the token be routed to no next Step (“dropped on the floor”) which is useful for Steps whose purpose is to monitor the workflow. If no Router returns a true value, the token is routed to a special dead-end Terminal Step.

The Step class also provides `start`, `stop`, and `recover` methods which it simply delegates to its Worker.

LabFlow

LabFlow provides methods for defining, executing, and managing a workflow. The workflow-definition methods let the programmer create a new workflow, and add Steps and Routers to it. There is also a method to define execution units and the number of processes desired for each. Finally, there is a method to define those aspects of the database schema that cannot be generated automatically from the Step definitions (mostly LabBase tag and type definitions), and a method to initialize a database using this schema. The principal workflow-execution method is `execute`. This method takes an execution unit as a parameter. The method runs forever, invoking the `execute` method of each Step in its execution unit, sleeping briefly when there is no work to be done. LabFlow also implements `start`, `stop`, and `recover` methods which it simply delegates to its Steps. The principal workflow-management method is `manage`. This method forks processes for each execution unit and causes each to run `start` then `execute`. This method makes an effort to determine if the previous shutdown of the system was clean. If not, it runs `recover` before forking the processes. The method watches the forked processes and creates replacements should any fail. There is also a `shutdown` method, generally invoked in response to a Unix `kill`, that attempts an orderly shutdown of the forked processes.

A SubFlow has a `terminate` method, generally called by the Worker of each Terminal Step, that decides whether the execution is complete, and if so, produces a summary result for the parent workflow. The summary result is in exactly the same format as the output of a Worker, and is fed into the parent Step as if it were produced by a Worker.

Discussion

Workflow management is proven, useful technology for LIMS-construction, esp. for projects with complex multi-step protocols. Workflow management offers good facilities for coordinating the performance of related tasks, handling failures, and tracking the progress of samples as they move through the project.

Beneath its apparent simplicity, however, lurk a host of gnarly subtleties. Parallelism, sub-workflows, multi-input workflows, and grouping of samples complicate the workflow model. Persistence and distribution challenge the system-level design. The need to easily integrate components constrains the detailed design of the framework.

LabFlow addresses a reasonable subset of the hard workflow management problems. We are confident that LabFlow, like its predecessors, will prove useful in practice. It is clear, though, that much research remains to be done in this area.

Acknowledgments

This work was supported by NIH grant R01 HG01367-01.

References

- Anaxagoras 1997. *Anaxagoras Home Page*. <http://www.anaxagoras.com/>. Anaxagoras BV.
- Buneman, P., S. B. Davidson, G. Hillebrand and D. Suciu 1996. A Query Language and Optimization Techniques for Unstructured Data. *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, Montreal Quebec (June 1996).
- Chen, I.-M. A. and V. M. Markowitz 1995. An Overview of the Object-Protocol Model (OPM) and OPM Data Management Tools. *Information Systems* 20(5):393-418.
- Cichocki, A., A. S. Helal, M. Rusinkiewicz and D. Woelk 1998. *Workflow and Process Automation: Concepts and Technology*. Kluwer Academic Publishers.
- Cimarron 1998. *Cimarron Software Home Page*. <http://www.cimsoft.com/>. Cimarron Software, Inc.
- CPN 1998. *World of Petri Nets*. <http://www.daimi.aau.dk/~petrinet/>. CPN group, Department of Computer Science, University of Aarhus (February 1998).
- Davis, J., W. Du and M.-C. Shan 1995. OpenPM: An Enterprise Process Management System. *Data Engineering* 18(1):27-32 (March 1995).
- Dietrich, W. F., J. Miller, R. Steen, M. A. Merchant, D. Damron-Boles, Z. Husain, R. Dredge, M. J. Daly, K. A. Ingalls, T. J. O'Connor, C. A. Evans, M. M. DeAngelis, D. M. Levinson, L. Kruglyak, N. Goodman, N. G. Copeland, N. A. Jenkins, T. L. Hawkins, L. D. Stein, D. C. Page and E. S. Lander 1996. A Comprehensive Genetic Map of the Mouse Genome. *Nature* 380:149-152 (March 1996).
- Durbin, R. and J. T. Mieg 1991. *A C. elegans Database*. Documentation, code and data available from anonymous FTP servers at lirmm.lirmm.fr, cele.mrc-lmb.cam.ac.uk and ncbi.nlm.nih.gov.
- Fayad, M. and D. C. Schmidt 1997. Object-Oriented Application Frameworks. *Communications of the ACM* 40(10):32-28 (October 1997).
- Goodman, N., S. Rozen, A. G. Smith and L. D. Stein 1998. The LabBase System for Data Management in Large Scale Biology Research Laboratories. *CABIOS* submitted.
- Goodman, N., S. Rozen and L. D. Stein 1994. Building a Laboratory Information System Around a C++-based Object-Oriented DBMS. *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, Santiago de Chile, Chile, The Very Large Data Bases (VLDB) Endowment Inc.
- Goodman, N., S. Rozen and L. D. Stein 1998. Data and Workflow Management for Large Scale Biological Research. *Molecular Biology Databases*. S. I. Letovsky. Kluwer Academic Publishers.
- Hsu, M., Ed. 1995. *Data Engineering: Special Issue on Workflow Systems*. IEEE Computer Society.
- Hudson, T. J., L. D. Stein, S. S. Gerety, J. Ma, A. B. Castle, J. Silva, D. K. Slonim, R. Baptista, L. Kruglyak, S.-H. Xu, X. Hu, A. M. E. Colbert, C. Rosenberg, M. P. Reeve-Daly, S. Rozen, L. Hui, X. Wu, C. Vestergaard, K. M. Wilson, J. S. Bae, S. Maitra, S. Ganiatsas, C. A. Evans, M. M. DeAngelis, K. A. Ingalls, R. W. Nahf, J. Lloyd T. Horton, M. O. Anderson, A. J. Collymore, W. Ye, V. Kouyoumjian, I. S. Zemsteva, J. Tam, R. Devine, D. F. Courtney, M. T. Renaud, H. Nguyen, T. J. O'Connor, C. Fizames, S. Faure, G. Gyapay, C. Dib, J. Morissette, J. B. Orlin, B. W. Birren, N. Goodman, J. Weissenbach, T. L. Hawkins, S. Foote, D. C. Page and E. S. Lander. 1995. An STS-Based Map of the Human Genome. *Science* 270:1945-1955 (December 1995).
- Kappel, G., P. Lang, S. Rausch-Schott and W. Retschitzger 1995. Workflow Management Based on Objects, Rules, and Triggers. *Data Engineering* 18(1):11-18 (March 1995).
- LIMSource 1998. *LIMSource Web Site*. <http://www.limsource.com/>. Write Away Communications.
- Liskov, B., A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers and L. Shriram. 1996. Safe and Efficient Sharing of Persistent Objects in Thor. *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, Montreal Quebec.
- McGoveran, D. and C. J. Date 1992. *A Guide to Sybase and SQL Server*. Addison-Wesley.

McHugh, J., S. Abiteboul, R. Goldman, D. Quass and J. Widom 1997. Lore: A Database Management System for Semistructured Data. *SIGMOD Record* 26(3):54-66 (September 1997).

Mohan, C. 1998. *Exotica*.
<http://www.almaden.ibm.com/cs/exotica/>. IBM Almaden Research Center.

Murata, T. 1989. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE* 77(4):541-580 (April 1989).

Rozen, S., L. D. Stein and N. Goodman 1995. LabBase: A Database to Manage Laboratory Data in a Large-Scale Genome-Mapping Project. *IEEE Transactions on Engineering in Medicine and Biology* 14:702-709 (September 1995).

Shlaer, S., S. J. Mellor and S. Mellor 1992. *Object Life Cycles: Modeling the World in States*. Prentice-Hall.

Steen, R. 1997. *Genetic Maps of the Rat Genome*.
<http://www.genome.wi.mit.edu/rat/public/>. Whitehead Institute/MIT Center for Genome Research.

Stein, L., A. Marquis, E. Dredge, M. P. Reeve, M. Daly, S. Rozen and N. Goodman 1994. Splicing UNIX into a Genome Mapping Laboratory. *Proceedings of the USENIX Summer 1994 Technical Conference*, USENIX.

Stein, L. D. 1996. How Perl Saved The Human Genome Project. *The Perl Journal* 1(2):5-9 (Summer 1996).

Stein, L. D., S. Rozen and N. Goodman 1994. Managing Laboratory Workflow With LabBase. *Proceedings of the 1994 Conference on Computers in Medicine (Comp-Med94)*.

Wall, L., T. Christiansen and R. L. Schwartz 1996. *Programming Perl*. O'Reilly & Associates.

WARIA 1998. *Workflow And Reengineering International Association Home Page*. <http://www.waria.com/>. Workflow And Reengineering International Association (WARIA).

WfMC 1996. *Workflow Management Coalition*.
<http://www.aiim.org/wfmc/>. Workflow Management Coalition (WfMC) (October 1996).

WICGR 1997. *Genetic and Physical Maps of the Mouse Genome*. <http://www.genome.wi.mit.edu/cgi-bin/mouse/index>. Whitehead Institute/MIT Center for Genome Research.