Cellarium:
A Computational Biology Workflowing Environment

Michael Gaiman*
Department of Computer Science
Brown University

December 21, 2006

1 Introduction

Cellarium is a Computational Biology Workflowing Environment. It is designed to simplify the creation and refinement of workflows. There are a large variety of bioinformatic disparate components: data sources, algorithms, visualizations, etc.. Cellarium provides a means of leveraging any such component, providing greater interoperability of components than ad-hoc approaches. Cellarium also provides a means of workflow composition. Many scientists are not programmers, yet need to develop complex bioinformatic workflows. Cellarium provides greater ease of use for technical and non-technical users alike.

Cellarium takes a programming language approach to workflows, with the guiding goals of interoperability and user-guided program composition. Specifically, Cellarium consists of the following components:

- A meta programming language, in which programs can be written that connect bioinformatic data sources, algorithms and visualizations. The language in essence, is the "glue" which connects disparate bioinformatic components into unified workflows. The language is presented in Section 2.
- A support environment around the programming language allowing non-technical users to compose workflows in our language. Users select data sources, choose which algorithms to perform on them and view visualizations (and possibly iterate stages), all without writing any code manually. The user environment is presented in Section 3.

This paper presents a programming language-based approach to Computational Biology workflows, briefly introducing the underlying programming language and then thoroughly discussing the support environment, the primary focus of my work.

2 Cellarium Meta Programming Language

In this section, we look at some examples of Bioinformatic components, and begin to develop an intuition for how they are combined to create workflows.

*Cellarium is work jointly developed with Eric Koskinen and Zach Shubert, under the direction of Sorin Istrail. In this paper I focus on the aspects of the system that I primarily designed and implemented.

2.1 Examples

Example 1: dot-plot

A common bioinformatic workflow involves the visualization of two sequences in a dot-plot. For example, a program may be written which opens two DNA sequences from FASTA source files[4], selects a limited range of bases, and produces a dot-plot as a result. Typically, such a workflow would be realized as a single, custom program. For example, some C code may extract the sequence from the FASTA file, use a library method to splice the sequences, and then use a graphics library to render the Dot Plot.

The pitfall with cusom programs is that there is a limited amount of reuse. If instead, a user wants to perform alignment on the sequences, they need to modify the source code and again devise a custom workflow. Within a domain, code may be modularized, however, subsequent programs must be written in the same language and become mired with the details of that language.

Cellarium's approach is to represent the *essence of a workflow* in a language specifically designed for Computational Biology workflows. For example, consider the following code:

```
(visualize dotplot
  (algorithm splice 583955, 853295,
    (data-source fasta ‘‘seq1.fasta’’))
  (algorithm splice 402595, 284924,
    (data-source fasta ‘‘seq2.fasta’’))
)
```

Three comments accompany this first example. Most fundamentally, we've separated the *composition* of the workflow from the *implementation* of the individual algorithms. Secondly, in this language, the stages of the workflow become explicit. Whereas custom programs have no formal barrier between stages, here each stage is a single expression in our language, and thus easily identified. Finally, the simplicity of the workflow written in this way makes it particularly ammenable to refinement. If, for example, a user wanted to switch to different ranges or even swap in alternate algorithms, it's easily accomplished without a complete program rewrite.

Example 2: Sequences

We now consider another program, not unlike the previous:

```
(visualize dotplot
  (algorithm dna-to-protein
    (data-source fasta ‘‘seq1.fasta’’))
  (algorithm dna-to-protein
    (data-source fasta ‘‘seq2.fasta’’))
)
```

Notice that the only change was the algorithm used. Rather than splicing the sequences, they are converted to protein sequences and then visualized as a dot-plot. If this had been written using custom code, such a seemingly simple change could be drastically involved. Firstly, it's unclear that the custom dot-plot code would be able to handle protein sequences *in addition to* DNA sequences. Secondly, the fact that the `dna-to-protein` algorithm could directly replace the `splice` algorithm highlights the fact that both algorithms agree on the types that they each consume. The simplicity of a change such as this in our language is an artifact of the sophistication of our type system which leverages the OCaml type system[1].

Example 3: Multiple Visualizations

The previous workflows have been fairly simple. Consider the following, slightly more complex workflow:

```
(let s1 (data-source fasta ‘‘seq1.fasta’’)
(let s2 (data-source fasta ‘‘seq2.fasta’’)
   (begin
      (visualize dotplot
          (algorithm splice s1 24824 34828)
          (algorithm splice s2 35833 45782))
      (visualize dotplot
          (algorithm splice s1 24824 34828)
          (algorithm splice s2 59834 67492))
)))
```

This example illustrates the need for two additional language features, which are essential to building complex workflows. Firstly, the `let` keyword allows you to bind a name to a data structure. This is particularly useful when the data structure is used multiple times (e.g. `s1`). Secondly, notice that there are multiple visualizations. Since visualizations consume data structures and return $Unit$, users must be able to allow computation to continue beyond a visualization. The primitive `begin` is a sequential composition operator, which allows users to chain multiple expressions together, with the return values being discarded.

Example 4: Interaction

The workflows shown so far are fairly static: visualizations produce outputs, but there is no way to accept input back from the user.

```
(visualize textual-results
    (algorithm global-align
        (interact select-region
            (visualize dotplot
                (data-source fasta ‘‘seq2.fasta’’ )
                (data-source fasta ‘‘seq2.fasta’’ )
))))
```

This example illustrates the use of an additional languagae primitive, `interact`. Interact is an input primitive: it awaits input from the user. Typically, `visualize` preceeds `interact`: first the user is presented with some data, and then they are prompted for input. In the above example, the `interact` allows the user to select a region of the dotplot to be globally aligned.

2.2 Language Primitives

In the previous section, several examples of the Cellarium Meta Programming Language (CMPL) were presented. This section discusses several of the language primitives in greater detail.

CMPL is composed of four primary language constructs and several supporting constructs. The primary constructs are

- `data-source` (`src`): Load data from a file or some other external source through an I/O operation. This is the primary means by which data is loaded into a workflow.
- `algorithm` (`alg`): Invoke the specifed algorithm with the given parameters.

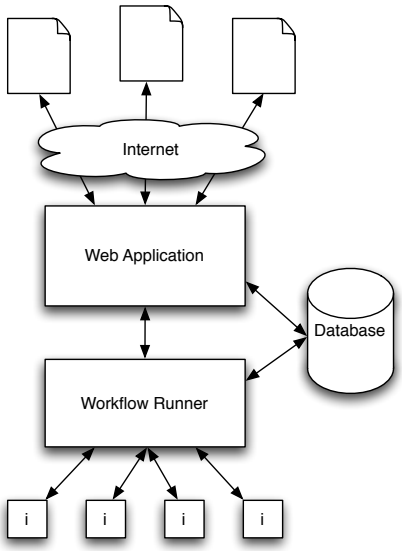

Figure 1: is an illustration of the Cellarium architecture. The Web Application is the central controller. It takes workflow submissions and dispatches them to the Workflow Runner where they are executed on one of a number of interpreters. Results are returned to the Web Application and shown to the clients.

- `visualize` (`vis`): Generates output destined for the user, generally in a stylized form (e.g. an image in the case of a dot-plot)
- `interact` (`ict`): Collects input from the user, typically in response to a previously generated visualization. This enables user-driven refinement and continuation.

In all of the above cases, some third-party code is invoked. For example, the dot-plot algorithm *is not part of the CMPL* language. Instead, it is an external algorithm which is invoked by the CMPL interpreter.

In addition to these primary primitives, the language also includes several standard language features. For example, CMPL includes a sequential composition primitive as well as a binding mechanism, which improves the practicality of programming in CMPL.

3 Cellarium Web Application

While the CMPL language is indeed simple to use, it is still a programming language and may be difficult for non-programmers to learn and use effectively. For this reason the language is encapsulated in a web application support environment. The CMPL language is abstracted away behind a visual composer and results viewer. Making Cellarium a web application has numerous benefits. It greatly eases installation and usage requirements. Users need not worry about providing reasonable compute resources. The user-facing portion of the Web Application consists of two major components, the Workflow Composer and the Result Viewer & Refiner, together with some support systems such as user registration and maintenance. Figure 1 is an illustration of the Cellarium architecture.

3.1 User Environment

The Cellarium Web Application is composed of several distinct components. This section details those components that are user-facing.

3.1.1 Workflow Composer

The Composer allows users to quickly design workflows and submit them for execution. The GUI, shown in Figures 2 & 3, consists of several components. The main component is the node view which draws the node tree. Clicking on a node causes it to be selected and brings up the Node Menu, a context sensitive floating menu that provides options for interaction. Possible options include the ability to insert parents, add children, delete nodes and also configure the currently selected node. Each construct in the CMPL language is represented as a specific node type:

- data-source (`src`): Represented by a single teal colored node.
- `algorithm` (`alg`): Represented by a single pale-blue colored node.
- `visualize` (`vis`): Represented by a single green colored node.
- `interact` (`ict`): Represented by a single purple colored node.
- `binding` (`let`): Binding is a more complicated node structure. Binding causes subworkflows to be defined. These subworkflows, which consist of the bound value, are drawn to the right of the main workflow and are given a numerical designation. Bindings may be used in the main system by adding an ID node, which is represented as a single pale-gray colored node. Figure 4 shows an example of a subworkflow.
- `sequential-composition` (`begin`): Represented by a single pale-gray colored node, labeled "sequentially-do".

The implementation details of the Cellarium Workflow Composer are detailed in Section 4.1.

3.1.2 Viewing Results

When the execution of the workflow has completed, the user is notified and given the location of the results. Generally, results are visualizations such as still images, text, video or even interactive mini-web applications designed to further explore the result-space. Results are organized by visualization and provide refinement options where relevant. Interactions are also handled through the Result Viewer.

3.2 Workflow Runner

Once a workflow has been submitted by the user, the web application performs a series of steps to generate the results. These steps are managed by the Workflow Runner, detailed in Section 4.2. Results are then made ready for viewing at the user's convenience.

4 Implementation

We implemented a prototype web application using the TurboGears rapid application development platform[2] and the PostgreSQL Database[5]. The composer and visualizer are AJAX-rich web pages, built upon the TurboGears framework. This section discusses the implementation details of both the composer and the workflow runner.

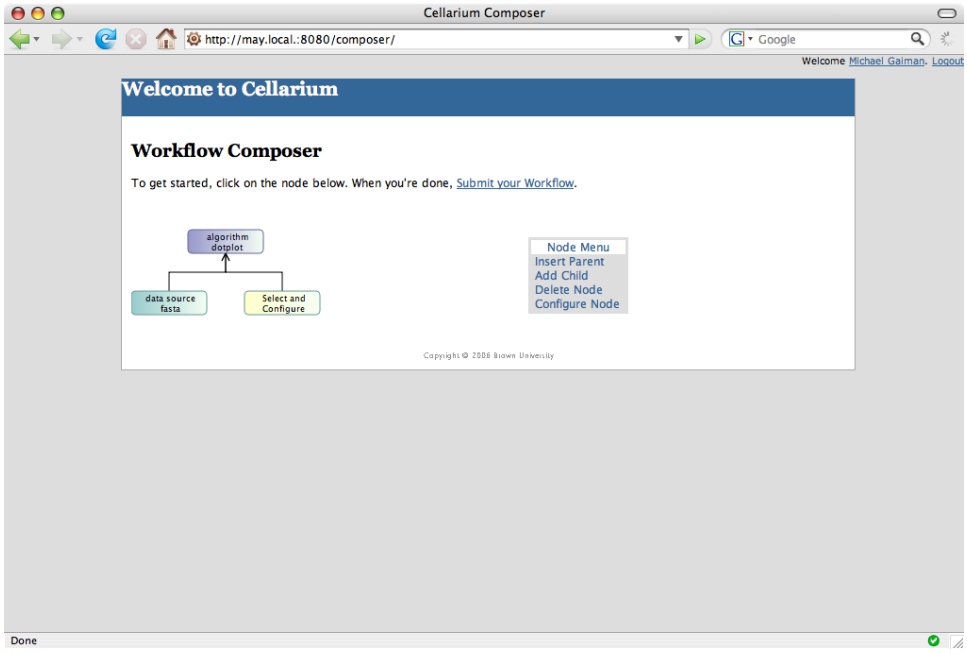

Figure 2: A view of the Cellarium Workflow Composer. In this example, a dot-plot is being constructed. On the left side, the nodes are viewed in a tree-like format. The Node Contextual Menu is located on the right side. It provides options that change depending on the currently selected node.

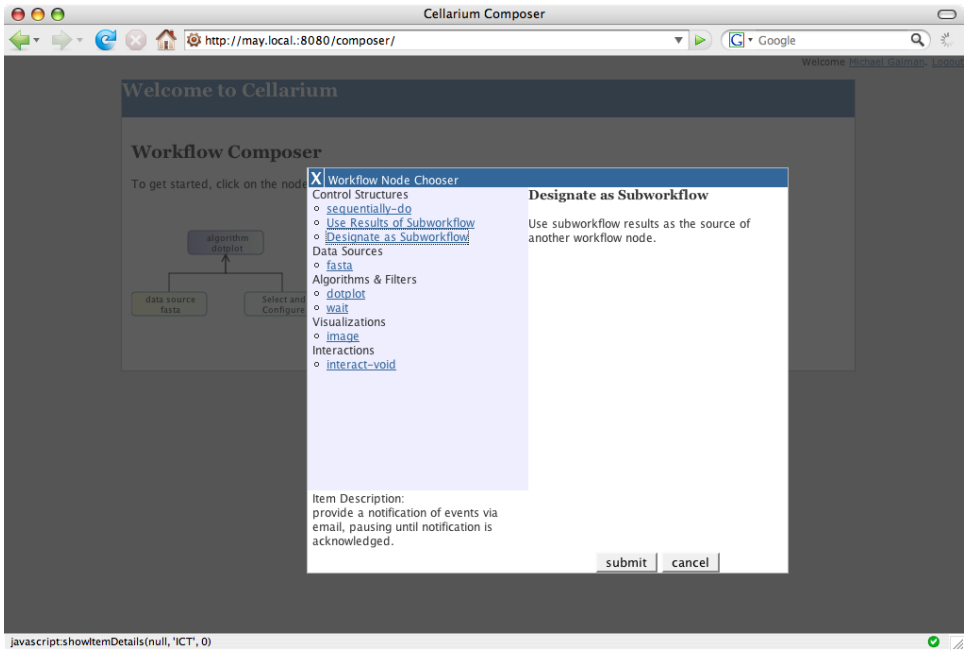

Figure 3: Workflow Composer showing the Node Chooser dialog. This example shows the node configuration dialog which allows users to choose from all valid options to determine what a node should do. The Node Chooser operates in a modal fashion.

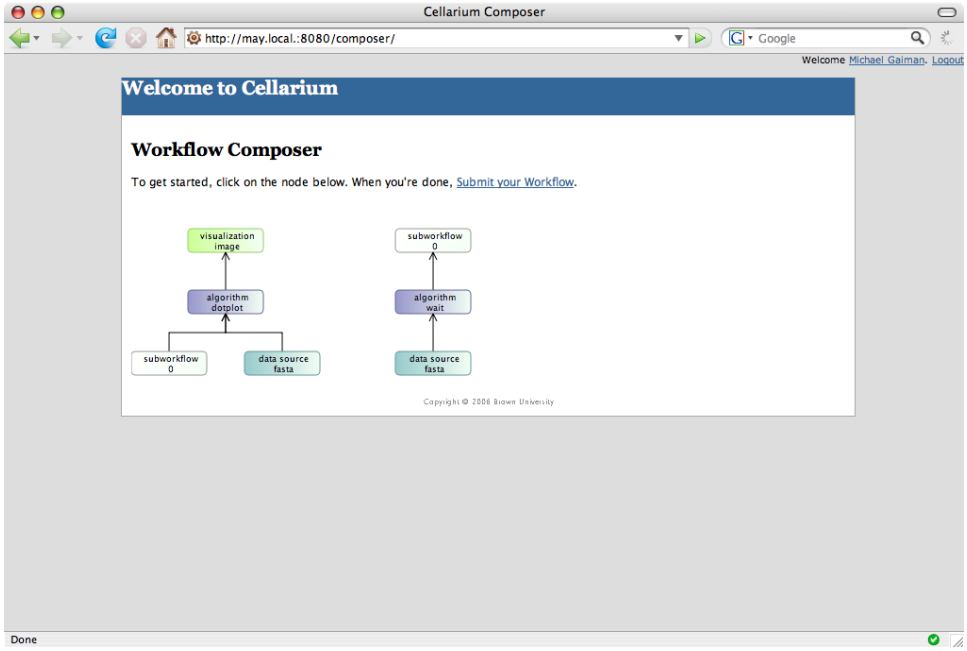

Figure 4: Cellarium Composer showing a workflow with a subworkflow. Subworkflows are the visual representation of the CMPL binding (`let`) construct.

4.1 Cellarium Composer

The Cellarium Composer has been implemented in standard AJAX fashion–using a mix of HTML and Javascript. The Composer relies on a number of support packages for various components. The node tree visual component is based on the ECOTree[6] package by Emilio Lobato. For Cellarium, it has been modified to dynamically resize and to broadcast events (among other things). Modal dialog support is provided via a modified copy of LightBox[3] which has been generalized to support non-image content.

Internally, Nodes are represented using a Javascript object structure. A superclass node is defined that provides implementations of several methods, key to which are the `cmpl()` and `addViewNode()` methods, which output the core CMPL source code and add the node to the tree view, respectively. Subclasses are created for each language primitive. These subclasses extend the superclass node to provide specific support for that language construct.

4.2 Workflow Running Implementation

Once a visual workflow is received from the user, it is translated into a full CMPL program that is then interpreted by the CMPL interpreter. The remainder of this section details how this hand-off happens.

Firstly, the workflow is added to the database by the web application, keyed to the submitting user. After adding it to the database, the Workflow Runner is notified that a new workflow needs to be ran. The Workflow Runner controls the processing of workflows. Currently the Workflow Runner is implemented as a simple fixed pool of worker threads, but the interface between the Runner and the rest of the web application has been defined to allow the Runner to be replaced by a compute-cluster solution if warranted in the future.

As the workflows are executed, results are stored in a directory on the web server. The directory

is unique to each completed workflow and is further subdivided by visualization.

The current state of the workflow execution is stored in the database. Possible states include QUEUED which is the initial state, it signifies that the workflow is waiting to be ran; RUNNING, the state in which the workflow is currently being executed; ERROR, which signifies that there was an error during execution. This could be encountered if the workflow was not a valid CMPL program or if it did not type-check, for example; WAITING, this state signifies that the runner is waiting for user interaction. Finally, if the execution completes, the state is set to DONE.

5 Future Work

The current version of Cellarium can be considered a prototype. There are several sets of enhancements required to make it production ready. This section enumerates these enhancements.

5.1 CMPL Enhancements

While we have sought to make CMPL well-rounded while staying true to its core purpose of being a light-weight glue language, there are several enhancements to the language that would make it more practical for daily use. The main limitation is the lack of iteration support. There is no looping mechanism. This can be worked around in the current system by moving the iteration into one of the native components, but this is sub-optimal. Cases such as repeatedly processing and then visualizing specific subsections of genes are difficult to describe in the current language.

While we have designed an XML transport layer to let various black-box and commercial components work within the system, the transport layer has not been implemented in the current interpreter. The current SWIG-based approach is cumbersome and does not provide as much abstraction as we would like.

5.2 Composer Enhancements

The workflow composer seeks to provide an easy to use, guided workflow composing experience. In order to conform to that mission, several enhancements are required. The current composer does not have much intelligence when it comes to replacing mid-level components. Currently whole subtrees have the possibility of getting replaced when switching to a different algorithm. While this is the safest course from a workflow validation point of view, it may lead to a compromised user experience.

A true drag-and-drop environment might prove to be easier to use than the current select-and-configure environment. Users would have the ability to choose components and drag them to places in the workflow. This would require a significant re-architecting of the tree layout component.

Currently, the composer takes a control-flow centric view of workflowing. A more natural approach, from a user perspective, might be to be data-flow centric. Usability studies would have to be conducted to determine which approach is the easiest to learn and to use effectively.

In addition, the current Results Viewer can be considered temporary. A permanent results viewer would allow users to interact with the workflow previously composed to view results and to create and submit refined workflows.

5.3 Workflow Runner Enhancements

The current Workflow Runner is a fixed thread consumer system. We have always envisioned the production system to be clustered. The current system could be expanded to be clustered with

little change to the interface between the core Web Application and the Workflow Runner. An intermediate solution would be to use a dynamic thread pool approach.

References

[1] Luca Cardelli. Basic polymorphic typechecking. *Sci. Comput. Program.*, 8(2):147–172, 1987.

[2] Kevin Dangoor. TURBOGEARS. `http://www.turbogears.org/`.

[3] Lokesh Dhakar. Lightbox js. `http://www.huddletogether.com/projects/lightbox/`.

[4] FASTA format description. `http://www.ncbi.nlm.nih.gov/blast/fasta.shtml`.

[5] PostgreSQL Global Development Group. PostgreSQL. `http://www.postgresql.org/`.

[6] Emilio Lobato. ECOTree. `http://www.codeproject.com/jscript/graphic_javascript_tree.asp`.