Wildfire User Manual
Version 2.0

Francis Tang
Bioinformatics Institute
A-STAR, Singapore

June 15, 2005

http://wildfire.bii.a-star.edu.sg

Revision: 1.13 – Date: 2005/06/13 02:58:58 Last Author: francis

Contents

1 Introduction **3**

2 Obtaining and Installing Wildfire **3**
2.1 Installing Wildfire 4
2.1.1 Windows installation instructions 4
2.1.2 Linux installation instructions 4
2.1.3 Uninstalling Wildfire 4
2.2 Installing GEL 4
2.2.1 Uninstalling GEL 5

3 Getting started: a tutorial **5**
3.1 A very simple workflow – multiple sequence alignment 6
3.1.1 Creating the workflow 6
3.1.2 Running the workflow 10
3.1.3 Running the workflow on a remote server 11
3.1.4 Overview 13
3.2 A short pipeline 13
3.2.1 Building the pipeline 13
3.2.2 Generalising the pipeline 14
3.2.3 Overview 24

4 Composing workflows **24**
4.1 Adding custom programs 25
4.2 Workflow constructs 28
4.2.1 Program instances 28
4.2.2 Dependencies (arrows) 29
4.2.3 Parallel container 29
4.2.4 Parallel foreach container 30
4.2.5 Parallel for container 30
4.2.6 While loop 31
4.2.7 Conditional branches 32

5 Executing workflows **32**
5.1 Export workflow as GEL script 32
5.2 Illegal workflows 33
5.3 GEL target platforms 33

6 Examples **36**

1 Introduction

There appear to be two trends in bioinformatics:

1. analyses are increasing in complexity, often requiring several applications to be run as a workflow, on large data sets; and

2. multiple CPU clusters and Grids are available to more scientists.

A consequence of the first observation is that these analyses require a large amount of processor time. Clearly the second observation has the potential to address this consequence.

The traditional solution to the problem of running workflows across multiple CPUs required programming, often in a scripting language such as perl. However, perl programming places such solutions beyond the reach of many bioinformatics consumers.

Wildfire addresses this problem, namely to let the bioinformatics consumer run analysis workflows on multiple CPU computers. Wildfire provides a graphical user interface for constructing and running workflows. Wildfire simplifies the interfaces of individual bioinformatics applications using dialog boxes with drop-down lists and check boxes. It also provides a drag-and-drop interface which allows for an intuitive way to compose these applications into a workflow, such as a pipeline. Wildfire is preconfigured to work with EMBOSS applications.

Once the workflow is constructed, Wildfire can execute the workflow (with the help of GEL) on a variety of target platforms including standard laptops/PCs, Beowulf-class clusters and Grids. Wildfire presents a uniform user interface to these different target platforms.

2 Obtaining and Installing Wildfire

To use Wildfire, you will need to obtain and install both Wildfire and GEL. Both are available from `http://wildfire.bii.a-star.edu.sg`.

Wildfire is the client program and the one which you will work with directly. You should install this on your personal computer, e.g. laptop or desktop.

GEL does the "back end work" and can be installed on the same computer (if it is running a unix-like operating system such as linux) or on a server on the network.

Without GEL, you can create workflows but there would be no way to execute them.

2.1 Installing Wildfire

Wildfire can be downloaded from `http://wildfire.bii.a-star.edu.sg` and is licensed under the GPL terms, an open source license.

Wildfire is available either as a zip archive, or as a Windows executable installer (recommended for Windows users).

2.1.1 Windows installation instructions

Download the Windows Installer `wildfire-2.0-w32installer.exe`, run it and follow the on-screen instructions. The installer will add Wildfire to your start menu, place an icon on your desktop and copy the examples into `C:\Documents and Settings\`⟨*username*⟩`\project\examples`. Use either the start menu or the desktop icon to start Wildfire.

2.1.2 Linux installation instructions

To install from the ZIP archive, just unpack the contents into a directory, e.g. `$HOME/wildfire-2.0`, and then from this directory, run the `install.sh` script. Thereafter, you can run Wildfire by executing `wildfire.sh` in this directory.

```
% cd $HOME
% unzip wildfire-2.0.zip
% cd wildfire-2.0
% ./install.sh
% ./wildfire.sh
```

The install script creates a directory called `$HOME/.gel`

2.1.3 Uninstalling Wildfire

To uninstall Wildfire if you had used the Windows installer to install, just run the Wildfire Uninstall program, or use the Windows Add/Remove Programs feature. This will remove the Wildfire program directory, and also remove the directory `c:\Documents and Settings\`⟨*user*⟩`\.gel`.

If you had installed Wildfire from the ZIP archive, then simply remove the program directory, e.g. `$HOME/wildfire-2.0` in the example above, and `$HOME/.gel`.

2.2 Installing GEL

Only Unix-like environments are supported by GEL. (GEL has been tested on i386-linux, ia64-linux, alpha-OSF and sparc-SunOS.)

Download gel-2.0.tar.gz from wildfire.bii.a-star.edu.sg. GEL is available without fee under an A-STAR license which allows for non-profit research use. The precise terms of the license are available from wildfire.bii.a-star.edu.sg website.[1] Extract this archive into your home directory and run the install script mk-wrapper.sh:

```
% zcat gel-2.0.tar.gz | tar xf -
% cd gel-2.0
% ./mk-wrapper `pwd`
```

The last step is important.

Add $HOME/gel-2.0 to your PATH. To check that GEL has been installed, try running gel on the command line:

```
% gel
GEL (Version: 2.0)
(c) 2004, 2005, Bioinformatics Institute, A-STAR, Singapore
...
```

If you see the banner above (plus help on command-line options), then you have successfully installed GEL.

2.2.1 Uninstalling GEL

To uninstall GEL, just remove the program directory, e.g. remove $HOME/gel-2.0 in the example above.

3 Getting started: a tutorial

In this section, we will illustrate the steps to create a simple project using Wildfire.

Table 1 shows the default locations of where a project's files are stored. For example, if my login name is francis and project name is myproject, then the directory in Windows would be

```
C:\Documents and Settings\francis\project\myproject ,
```

and in linux would be

```
$HOME/project/myproject .
```

You will need to know these directories to add new files to the workflow.

[1] Please contact BII for information about other licenses.

OS	Default project directory
Windows	`C:\Documents and Settings\`⟨*user*⟩`\project\`⟨*name*⟩
Linux	`$HOME/project/`⟨*name*⟩

Table 1: Default directories for Wildfire projects

3.1 A very simple workflow – multiple sequence alignment

We will first start with a very simple workflow. This workflow runs one program (EMMA) which will compute a multiple alignment of a collection of sequences.

Note: you will need to make sure that ClustalW is installed and the binary `clustalw` is available on the PATH.

3.1.1 Creating the workflow

Start Wildfire. You will be presented with the main Wildfire window (Fig. 1). From the `File` menu, select `Open Project`. In the Open Project dialog box (Fig. 2) click on the "Create Folder" button and choose a suitable name for your project. Here we have chosen `myproj`. Once you have named your project, click "OK" to create the project. When you click "OK", Wildfire will create a number of default directories in your project directory. These are shown in Fig. 3. The purpose of these directories are explained in Tab. 2.

Directory	Purpose
`acddir`	Directory for storing ACDs of programs that are specific to this project
`execdir`	Directory for storing binaries and scripts for programs that are specific to this project
`ipdir`	Directory for storing read-only input files
`result`	Directory for storing the results of a project-run

Table 2: The meaning of default project subdirectories.

Our first workflow will consist of one program, the multiple sequence alignment program emma (which is an EMBOSS wrapper for ClustalW). In the left panel, select the EMBOSS tab, and then select the *template* "emma" by either using the hierarchical menus `ALIGNMENT -> MULTIPLE -> emma` (see Fig. 4), or the alphabetical list. Now you can click on an empty part of the project canvas (the main panel) and Wildfire will create the an emma *instance* in the workflow, which is drawn as a yellow box. The top half of the box is the name of the program, and the bottom half shows a unique serial number which is used to distinguish different instances of programs on the same canvas.

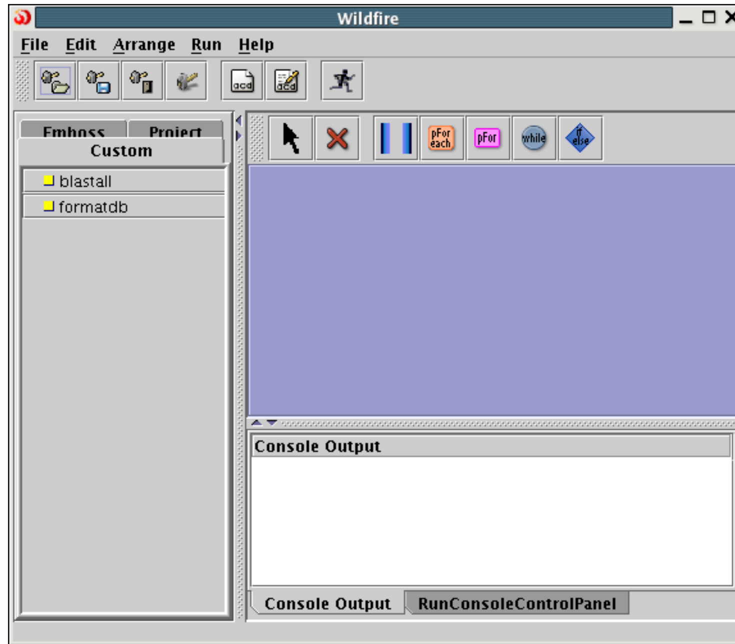

Figure 1: The Wildfire main window.

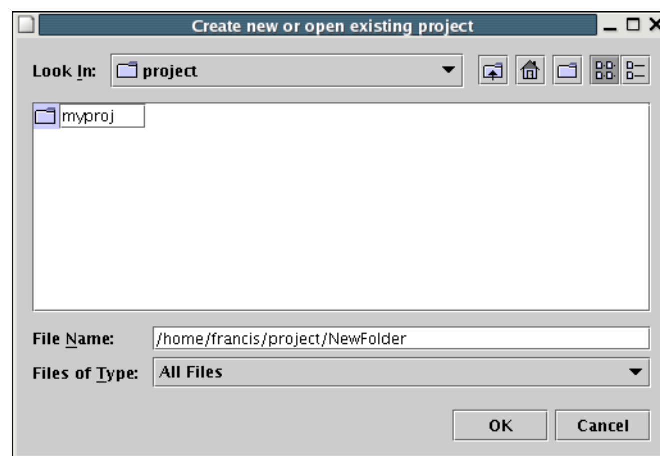

Figure 2: Dialog box for opening and creating new projects.

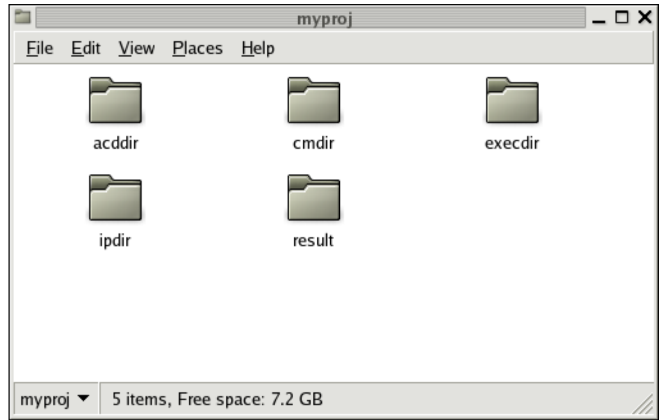

Figure 3: Standard project subdirectories.

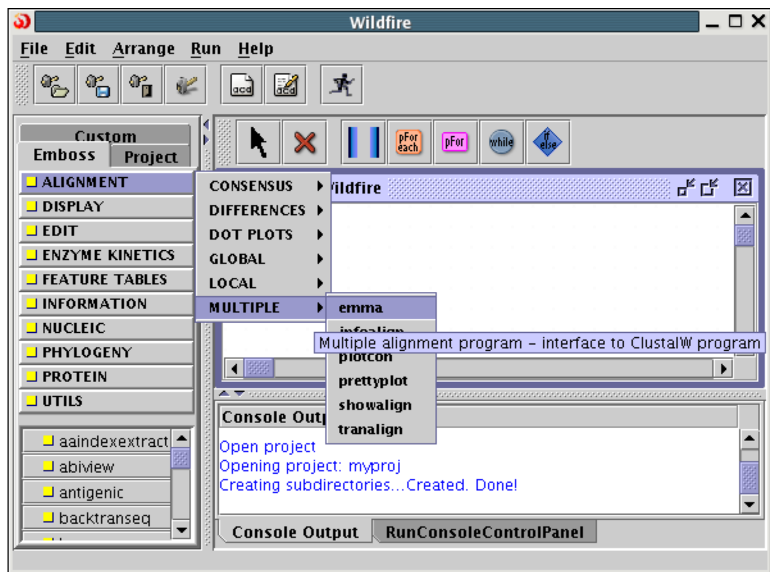

Figure 4: Selecting the EMMA template from the hierarchical EMBOSS menus.

Double-clicking on the instance, i.e. the yellow box, will bring up its properties menu (Fig. 5). From here you can control the options passed to EMMA for this instance.

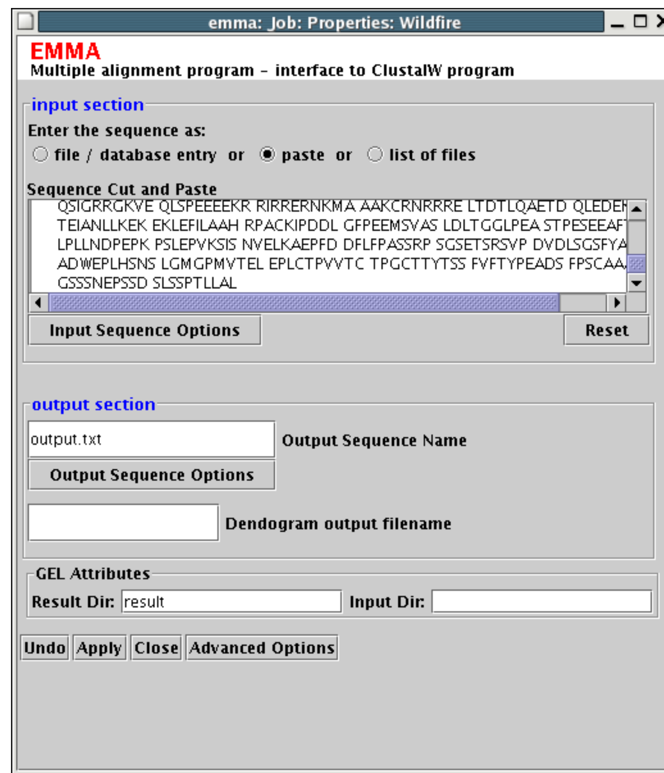

Figure 5: Changing the properties for the EMMA program.

- In the “input section”, select “paste”, and then in the “Sequence Cut and Paste” text box, paste some sequence data. (EMMA expects several sequences in FASTA format.)
- In the “output section”, type the name of the file in which the multiple alignment will be stored, for example `output.txt`.
- Finally, in “GEL Attributes”, type `result` in the the “Result Dir” text box. This instructs GEL to copy the output of the workflow into the directory `result`.

Now click the “Apply” button at the bottom to accept the changes and “Close” to close the dialog box.

Remember to save your workflow using the `File -> Save` menu.

3.1.2 Running the workflow

To run the workflow, select `Run -> Start Run`. This will bring up a dialog box with a list of "Profiles". For the moment, select profile "local" (which is available by default) and then click "Run".

The "local" profile has been configured to run the workflow on the same machine which is running Wildfire. Therefore, it is assumed that GEL is already installed on this machine (as are EMBOSS and ClustalW).

As Wildfire delegates execution to GEL, the canvas is updated to reflect the state of execution. Small icons will appear on the top right corner of the yellow boxes. A small cog indicates that the job has been queued for execution or has started running, and a green tick indicates that the job has completed. A dialog box will alert you when the whole workflow has finished running. After execution, you will see a screen like that of Fig. 6.

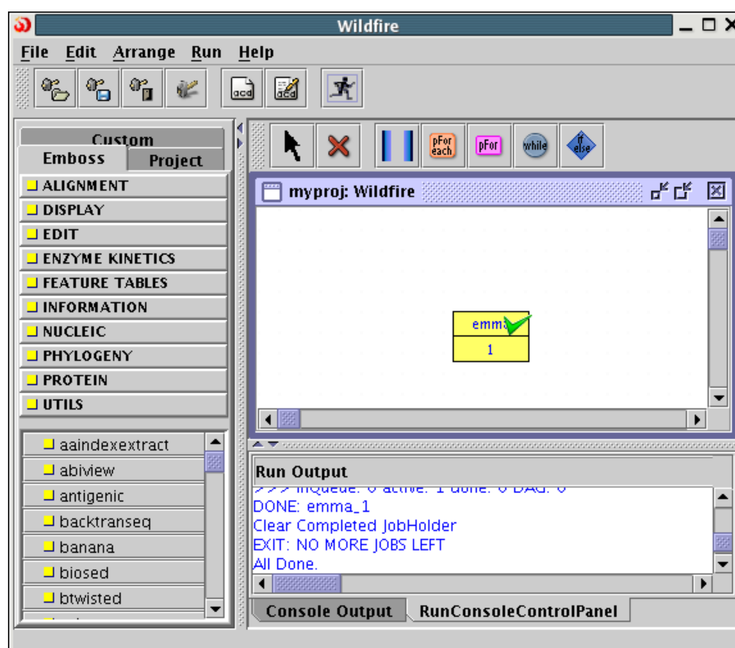

Figure 6: Wildfire window after workflow execution has completed.

To view the results, you can either browse the `result` subdirectory of the project directory (as shown in Tab.1), or use the project browser tab in the left pane. Figure 7 shows the project browser pane after execution.

Notes: If you had forgotten to specify a result directory, then there will be no files in the directory `result` when the workflow has finished. The output files will still be available in the temporary work directory which has the form `Jtmp-XXXX` where `XXXX` is typically a 10 digit number.

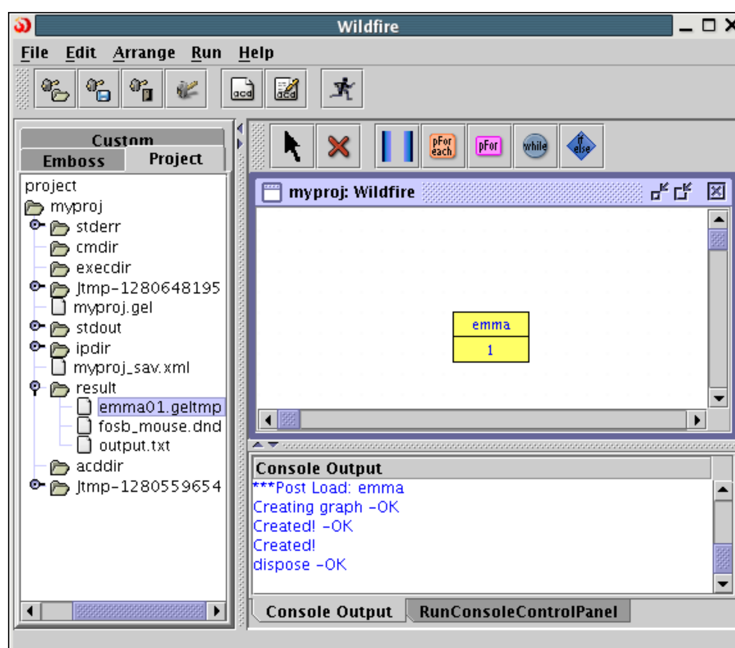

Figure 7: The project browsing pane. Text files and image files can be viewed by double-clicking on the file name.

3.1.3 Running the workflow on a remote server

We will now run the workflow on a remote server without change. You may want to do this because

1. Wildfire is running on Windows, and since GEL only runs on UNIX-style operating systems, you must run the workflow on a remote server.

2. The workflow takes too long to run your machine and you would like to take advantage of multiple CPUs on a remote server such as a cluster.

3. You don't have all the necessary applications installed on your machine, but they are available on a remote server. For example, this example requires (other than GEL) EMBOSS and ClustalW.

Fortunately, running the workflow on a remote machine is not much harder.

Recall in the previous section, to run the workflow we selected the "local" profile. To run the workflow on a remote server, we must create a new profile. In the "Start Run" dialog box, click on the "New" button to bring up the dialog box similar to that in Fig. 8.

The "Profile" field is just the name of the profile and can be anything you feel appropriate. Select the "Remote" radio button to indicate that this is a profile

for remote execution. In “Hostname”, give the internet host name of the remote server. Select “Local Threads” for “Scheduler”, which means that programs will run directly on the server itself without going through a scheduler. The “Local Threads” option will work for all servers. The “NumProc” field specifies the maximum number of concurrent programs when running the workflow; this value can be used to tweak the workflow execution performance, but it does not change the results. Click “Save” to save the profile.

The newly created profile will be available when you next want to run your workflow. To use the profile, just select it and click “Run”. Wildfire will ask for your login password for the remote server specified in the profile.

When running a workflow on a remote machine, Wildfire will copy over all the related files to the remote server, and then run GEL remotely. When execution is complete, Wildfire will copy the results back to the local machine. You can view the results using the project browser as before.

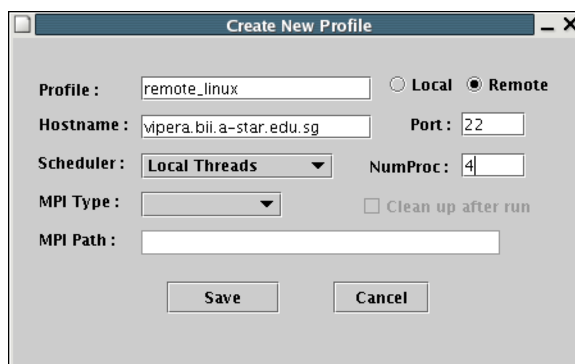

Figure 8: Dialog box for creating a new execution profile. Here the user can specify the name of the remote machine, and what sort of scheduler it uses. The “Local Threads” option means that this profile will not use any scheduler but rather will run the jobs directly, up to a maximum number of four (given in the NumProc field) concurrent threads.

Since this is a simple workflow, you will likely find that total execution time is longer (i.e. the time between clicking “Run” and receiving the “completed” notification is longer). This is because remote execution introduces further overhead which is required to move files around. However, for larger workflows which exhibit parallelism, multiple CPUs on the remote server will be able to run the workflow faster.

3.1.4 Overview

In this first section, we created a simple, one-program workflow. We then showed how to execute it on the local machine, and then remotely on a server. You will find that local and remote execution modes behave similarly from the point of view of result files.

3.2 A short pipeline

In this section, we'll create a short pipeline step by step.

For the sake of example, our pipeline will consist of two steps: `getorf` and `garnier`. EMBOSS program `getorf` finds all open reading frames (ORFs) in a genomic sequence, and returns them as peptide sequences. EMBOSS program `garnier` predicts protein secondary structures in these peptide sequences.

After creating the initial pipeline, we will modify it so that it can run concurrently on several input sequences.

3.2.1 Building the pipeline

First we create the `getorf` and `garnier` instances on the canvas, by selecting their templates from the EMBOSS tab on the left panel, and then clicking on the canvas. With the two instances created, the canvas should look like that in Fig. 9.

We would like to configure the `getorf` instance so that it reads a nucleic sequence from `seq01.fasta` and then writes the ORF sequences to `seq01.orf`. Similarly, we would like to configure `garnier` instance so that it reads the peptide sequences from `seq01.orf` and then writes the secondary structure predictions to `seq01.struct`. So for each instance, we double-click and fill-out the properties dialog box as shown in Figs. 10 and 11.

The workflow, as shown in Fig. 9, is not a pipeline since there is no dependency defined between the two instances. As it stands, `getorf` and `garnier` are independent, which means that the interpretor can run them in any order, or concurrently if necessary. However, this is not what we intended. Rather, we expect `garnier` to run only after `getorf` has completed. To express this in Wildfire, we have to draw an arrow between the two instances by cicking on the right edge of `getorf` and dragging the arrow to the left edge of `garnier`. The canvas should now look like that of Fig. 12.

Since `getorf` will run first, we add the directory name `ipdir` to the "Input Dir" field of the `getorf` properties dialog. This instructs GEL to first copy the files from the directory `ipdir` into the working directory before executing `getorf`. Similarly, we add the directory name `result` to the "Result Dir" field

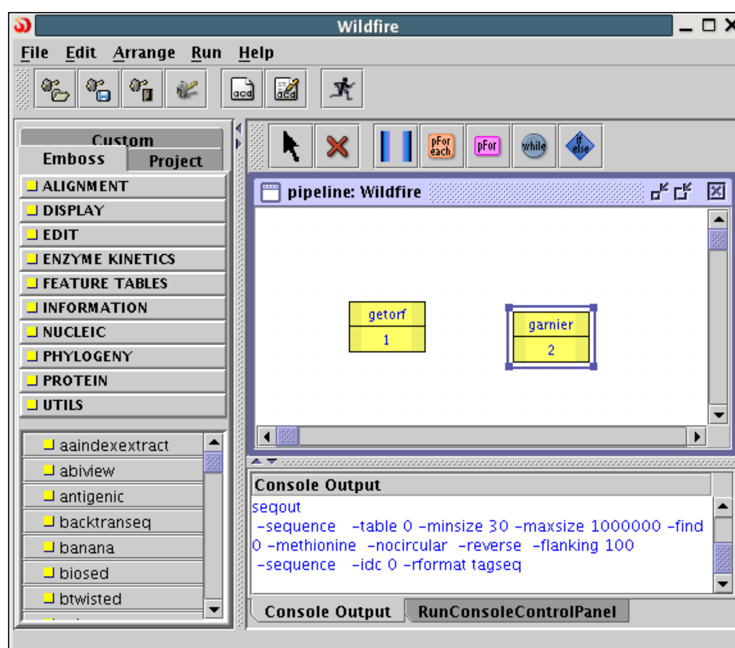

Figure 9: EMBOSS programs `getorf` and `garnier` on the Wildfire canvas.

of the `garnier` properties dialog; this instructs GEL to copy the contents of the working directory into the subdirectory `result` after `garnier` has completed.

The Wildfire workflow can be paraphrased as follows:

1. copy contents of `ipdir` into the working directory;
2. run `getorf` with input file `seq01.fasta` and write the output to `seq01.orf`;
3. run `garnier` with input file `seq01.orf` and write the output to `seq01.struct`; and,
4. finally, copy the new files (i.e. those that did not originate from `ipdir`) into the directory `result`.

Before you run the workflow, you should make sure that the input datafile `seq01.txt` can be found in the `ipdir` directory. You should copy the file there using the Windows Explorer or other file manager.

After execution of the pipeline, the Wildfire canvas and Project Browser pane should look like that of Fig. 13.

3.2.2 Generalising the pipeline

Now suppose you need to run this pipeline on a collection of independent input sequences, e.g. `seq01.fasta` and `seq02.fasta`. Wildfire has a workflow

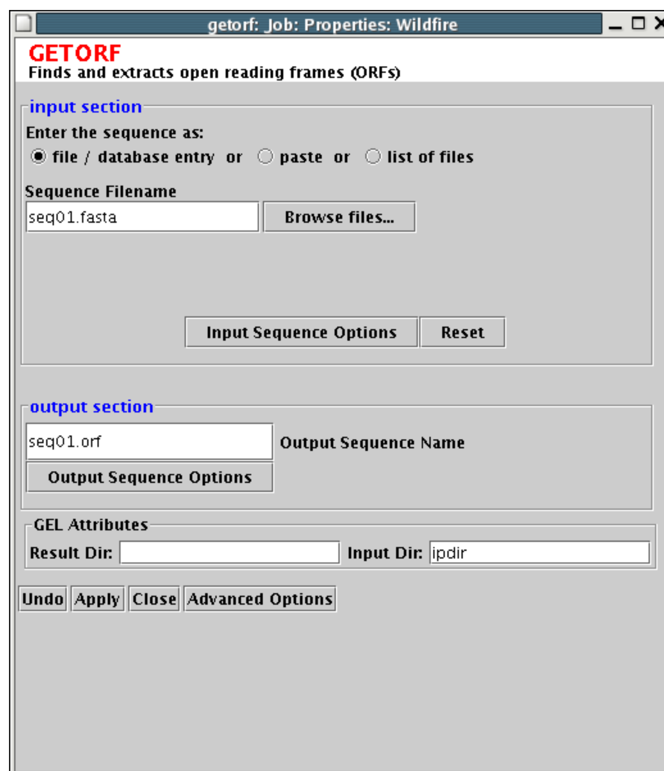

Figure 10: Properties for `getorf` instance.

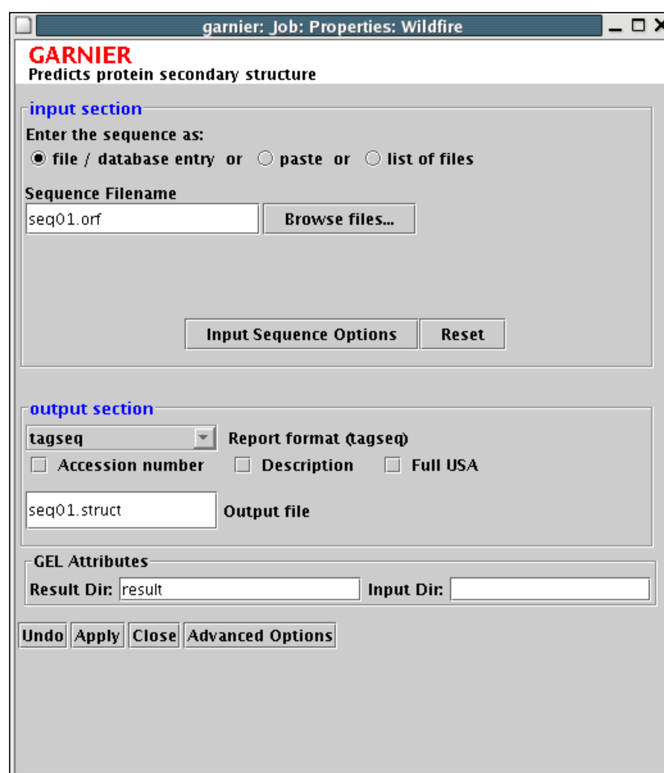

Figure 11: Properties for `garnier` instance.

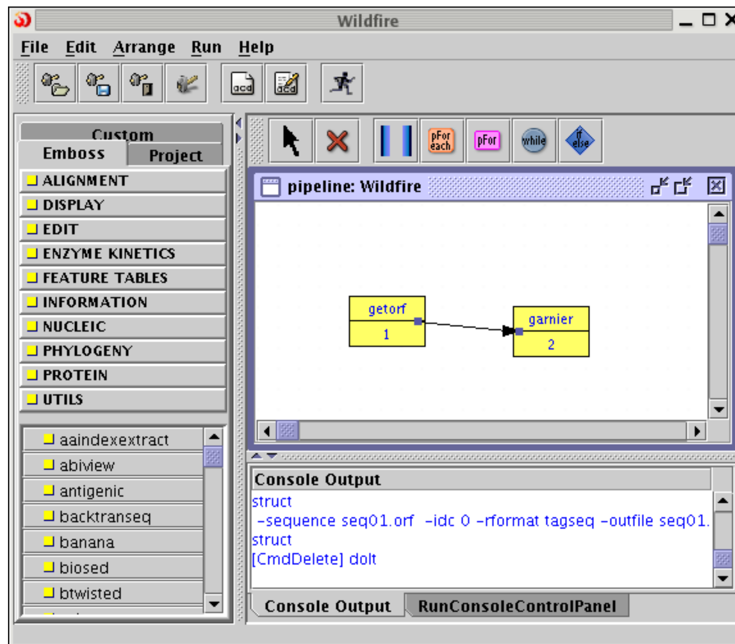

Figure 12: The final pipeline showing the dependency arrow between `getorf` and `garnier`.

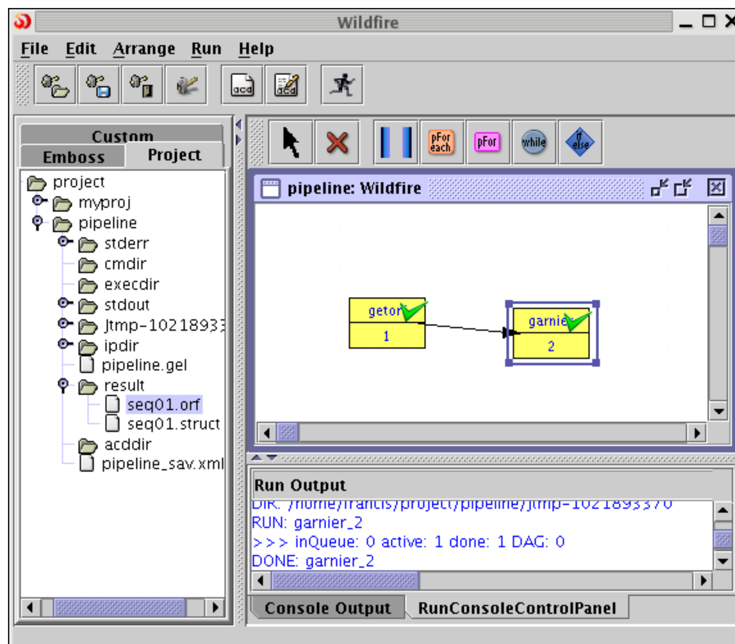

Figure 13: Wildfire showing the canvas and project browser pane after executing the pipeline.

construct which can do exactly this.

Starting from the previous workflow, click on the "pforeach" button, which is the fourth button from the left in the toolbar above the canvas. If you then click on the canvas, a "pforeach container" is drawn on the canvas. Resize this container so that it envelopes both `getorf` and `garnier`, as shown in Fig. 14. Double click on the banner of the pforeach container to bring up the dialog shown in Fig. 15, where we specify `$f` for the variable and `seq*.fasta` for the filename pattern. What this means is that for each file matching `seq*.fasta`, there will be a separate copy of the workflow inside the pforeach container for which the variable `$f` is interpreted to be the name of the file.

To give a concrete example, suppose the work directory contains two files: `seq01.fasta` and `seq02.fasta`. Both of these files match the pattern `seq*.fasta`, and so there will be two independent copies of the pipeline `getorf -> garnier`; in one copy, `$f` is understood to mean `seq01.fasta` and in the other, `$f` is understood to mean `seq02.fasta`.

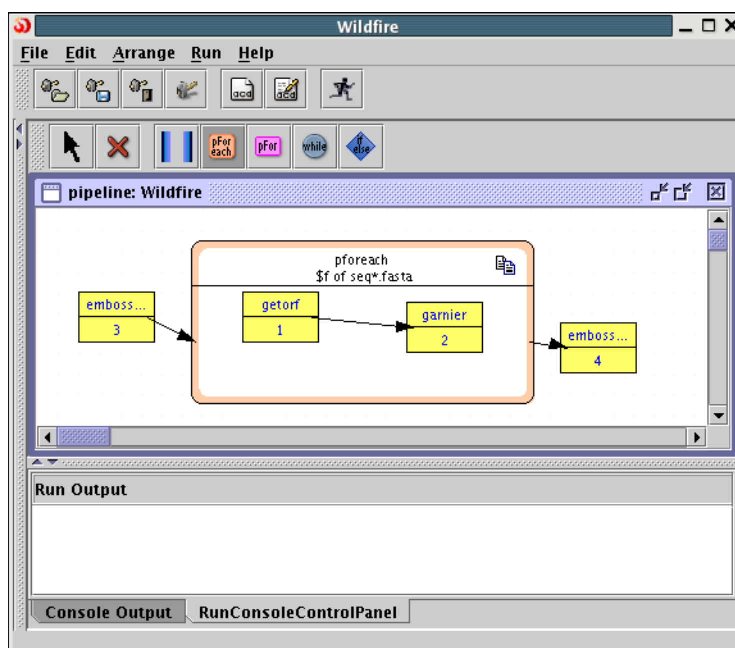

Figure 14: The modified workflow showing the pforeach container bounding `getorf` and `garnier`.

The workflow is not quite complete. We have to modify the properties of the `getorf` and `garnier` instances so that they make use of the variable `$f`. Figure 16 illustrates how to use variables. In the textbox in the input section, the equals symbol (=) means that what follows is an expression, not a literal value (cf. spreadsheet applications such as Microsoft Excel). So the expression

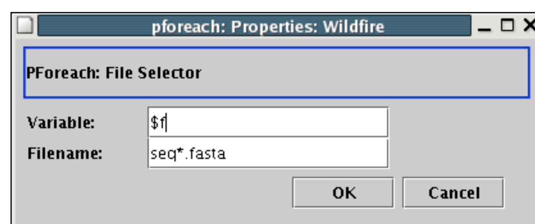

Figure 15: The properties dialog of the pforeach container. We choose to iterate over the variable `$f` over all files matching `seq*.fasta`.

"`= $f`" means "the value of variable `$f`". In the textbox in the output section, we see the concatenation operator, written as dot ("`.`"). The expression "`= $f . ".orf"`" means "the value of variable `$f` appended with the four characters `.orf`". Note that we no longer specify anything in the "Input Dir" field.

Similarly, Fig. 17 shows the properties of the `garnier` instance. Note that in the input section, we specify "`= $f . ".orf"`", i.e. the same as in the output section of the `getorf` instance. Finally, for the output section, we specify "`= $f . ".struct"`".

Returning to our concrete example, for one copy of the pipeline, `getorf` will read data from `seq01.fasta` and write the ORFs to `seq01.fasta.orf`. Then `garnier` will read the ORFs from `seq01.fasta.orf` and write the secondary structure predictions to `seq01.fasta.struct`. The second pipeline is similar, except the file names are `seq02.fasta` followed by `seq02.fasta.orf` before finally `seq02.fasta.struct`. Since the pforeach container creates *independent* copies of the pipeline, the programs can be executed concurrently, if possible.

If you are concerned that filenames such `seq01.fasta.orf` are inelegant and prefer `seq01.orf`, do not worry for Wildfire indeed has an operator which allows you to program it this way. We refer the reader to the next section for a more detailed explanation of this operator.

We also add two instances of `embossversion` which run before and after the pforeach component. The `embossversion` program performs no computation but the instances rather form a place holder to allow us to specify the Input and Result directories of this workflow. This is an implementation artefact of GEL rather than a conceptual feature. The properties of the first instance of `embossversion`, the one labelled "3", are set according to Fig. 18. Similarly, the properties of the second instance, namely that labelled "4", are set according to Fig. 19.

Construction of the generalised workflow is now complete. Now we can copy our input files into the `ipdir` directory (making sure to name them `seq`*XXX*`.fasta`),

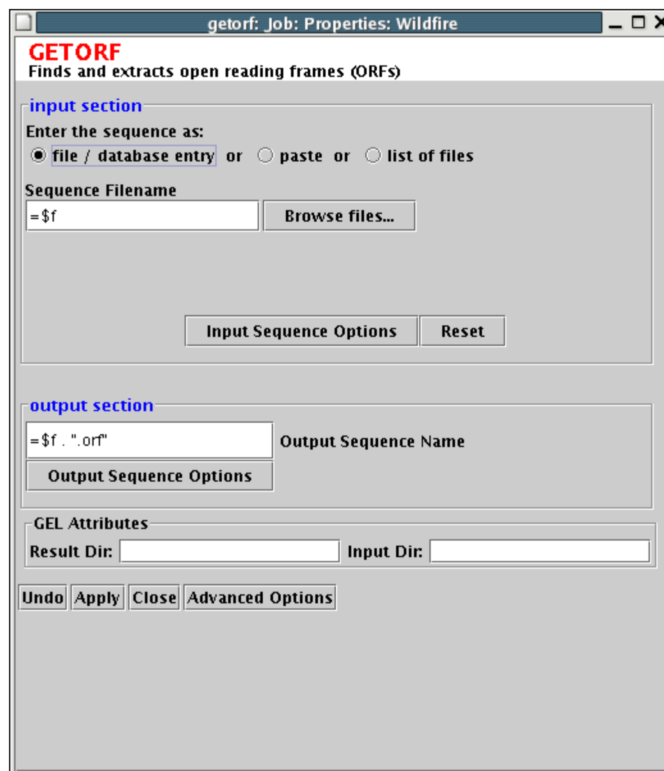

Figure 16: The properties dialog for the `getorf` instance in the generalised pipeline. Here we use the value of variable `$f` to make it generic.

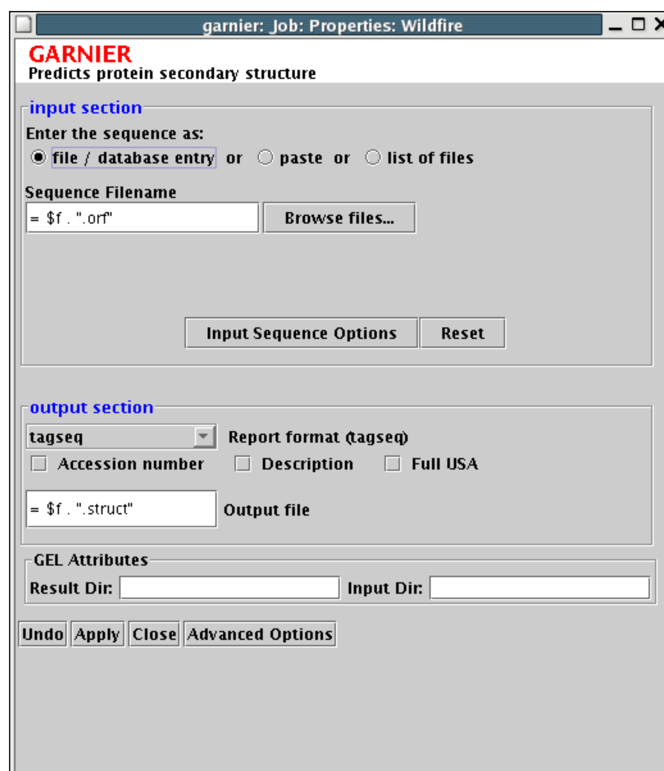

Figure 17: The properties dialog for the `garnier` instance in the generalised pipeline. Here we use the value of variable `$f` to make it generic.

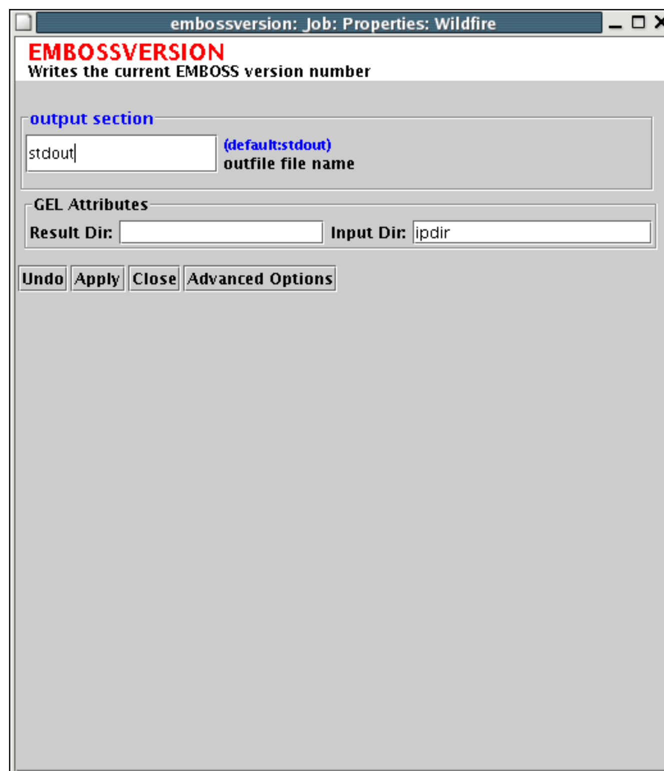

Figure 18: The properties of the `embossversion` instance labelled “3”. Here we set “Input Dir” to `ipdir`.

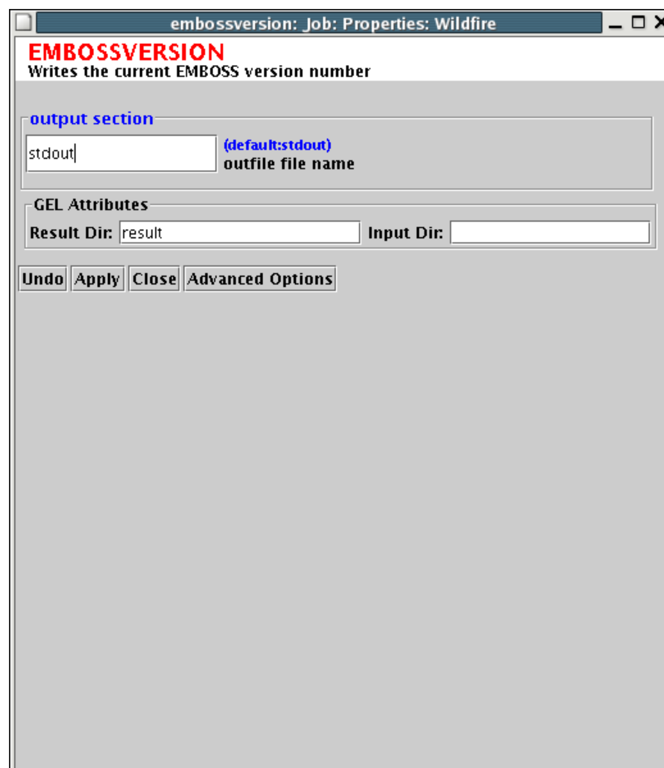

Figure 19: The properties of the `embossversion` instance labelled "4". Here we set "Results Dir" to `result`.

and run the workflow. Figure 20 shows the results pane after executing the workflow on two input files `seq01.fasta` and `seq02.fasta`.

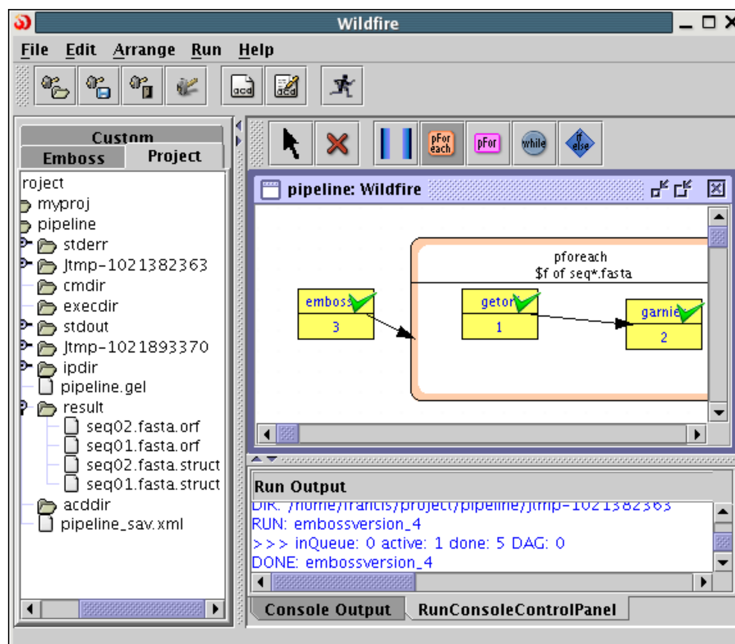

Figure 20: The project browser pane in Wildfire after executing the generalised pipeline on two input files.

3.2.3 Overview

We have now taken the orginal two-stage pipeline and generalised it to run on a collection of input files. The constructs used in this example create independent instances of the pipeline which can be executed concurrently if allowed by the target platform.

4 Composing workflows

This section introduces the Wildfire features used in the constuction of workflows. We first explain how to add custom, i.e. non-EMBOSS programs, to Wildfire. We then explain each of the Wildfire canvas constructs.

4.1 Adding custom programs

In the left pane of the Wildfire workspace, you will see several tabs. The tab named "EMBOSS" contains templates for all the EMBOSS applications. You use these templates to create instances on the Wildfire canvas.

The pane named "Custom" contains the templates for non-EMBOSS programs. We have already created templates for two NCBI BLAST applications: `formatdb` and `blastall`. You can immediately use these applications in your workflow, in the same way you would use EMBOSS applications. (It is assumed the target platform has both applications installed and available in the PATH.)

Each program in the EMBOSS and Custom panes has a corresponding description of its command-line parameters. In fact, Wildfire uses the EMBOSS ACD format for this description. To add a new program to the "Custom" panel, you must create a corresponding ACD file. Wildfire provides an ACD editor which allows you to do this.

To create a new ACD, select `Edit -> Edit Custom ACD -> New ACD`. This will bring up the ACD editor dialog box as shown in Fig. 21. The bottom half of the dialog box is used to define the command-line options for the new application. The fields in th etop half of the dialog box are explained below.

Application [Mandatory] This is the filename of the executable.

Shared ACD [Advanced] Select this checkbox if you want this definition to be shared by all projects. (Ticking this box will cause changes to this ACD to affect all project which use it. Therefore it should be used with care.)

Information Textual description of what this program does.

Architecture Architecture for which the executable has been compiled, in the case where the binary is stored with the project. In the case where the executable is to be found in the PATH, i.e. if the executable is already installed on the execution host, this field is ignored. Currently only the Condor executor supports this architecture heterogeneity.

Library This is reserved for future GEL interpretors which support software heterogeneity.

Remote Application Request This field is reserved for future use.

Directory (Local) This field specifies where the executable file is stored. If the executable file is already installed on the host computer and can be found in the PATH variable, then leave this field blank.

Hostname (Remote) This field is reserved for future use.

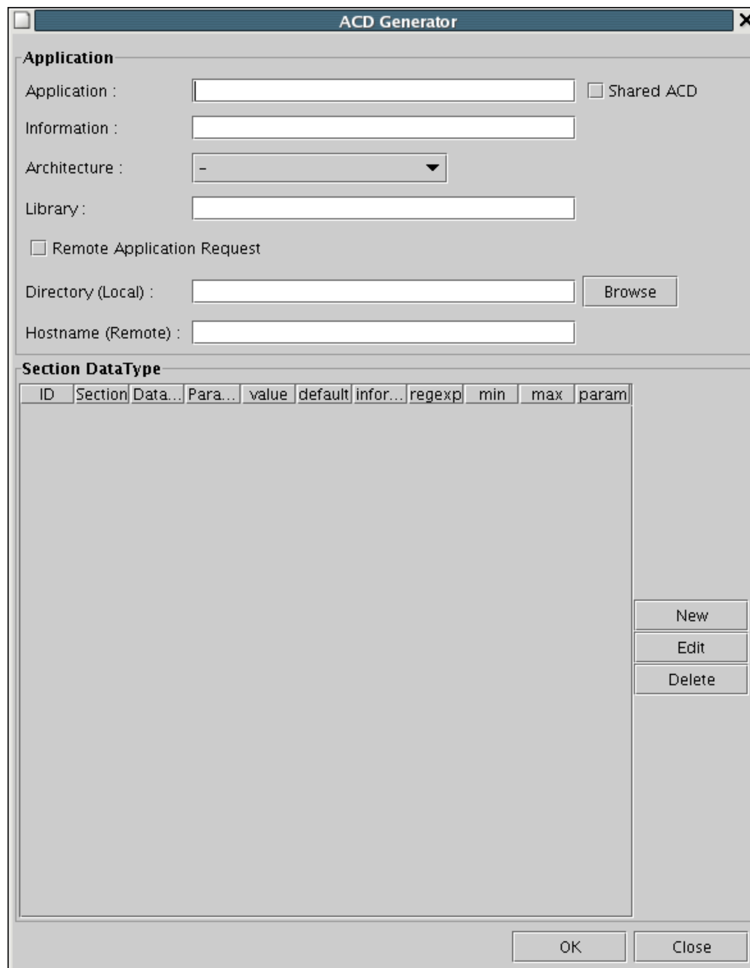

Figure 21: ACD editor dialog box

A new program can be used in a workflow using two schemes: (1) put the executable file together with the project files; or (2) install the executable on the execution host by putting it into the PATH. For the first option, you must set the "Directory (Local)" field to point to the executable. For the second option, you must leave "Directory (Local)" blank.

The syntax for the command-line parameters for the program must be defined using the bottom half of the dialog box. To define a new user-specifiable parameter, click on the "New" button to the right of the list of parameters. This will bring up the dialog box shown in Fig. 22.

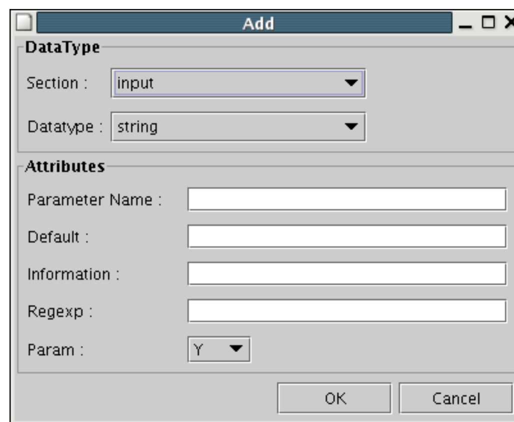

Figure 22: New parameter dialog box

The "Section" field simply defines the organisational structure of the resulting properties dialog box, i.e. whether the field should appear in the "Input", "Output", "Required" or "Advanced" sections. The "Datatype" field allows you specify what type the argument should take. This allows Wildfire to do some further checking of the command-line parameters, however, "string" will suffice for almost all applications if you are prepared to forego this extra level of checking.

The remaining fields are explained below.

Parameter Name This field gives the name of the parameter. For example, fi the parameter name is `xyz`, then Wildfire will add the command-line option `-xyz` *nnn* where *nnn* is the value given by the user. If the value *nnn* should be specified without a dash-option, e.g. `prog` *nnn*, then the `Param` option should be set to "N" (see below).

Default This is the default value for this parameter.

Information This is a help string.

Regexp This is a regular expression which further constrains allowable values.

Param If this is set to "Y", then the the command is given with a dash, the *Parameter Name*, space and then the value. For example `prog -xyz` *nnn*. If this is set to "N", then argument is passed without the *Parameter Name*, e.g. `prog` *nnn*.

The "list" datatype allows you to specify a list of possible values. In this case, the "Value" field should be given in the form `val:display` pairs separated by semicolons. For example, `Y:Protein; N:Nucleotide` will display a drop-down list with options "Protein" and "Nucleotide", and selecting them would give values "Y" and "N" respectively.

A similar datatype is "select". For this data type, "Value" field should be a semicolon-delimited list of names, and they will be numbered sequentially from 1. For example, the value `one; two; three` would display a drop-down list with options "one", "two" and "three". The corresponding values to these options would be `1`, `2` and `3`.

Similarly, to edit an ACD file, select `Edit -> Edit Custom ACD -> Edit ACD`. This will bring up the dialog box shown in Fig. 23.

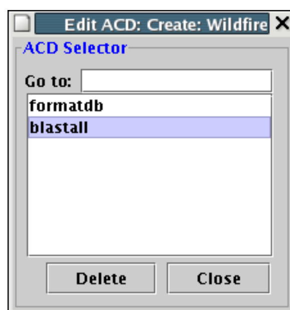

Figure 23: ACD edit dialog box

4.2 Workflow constructs

4.2.1 Program instances

Program instances are the atomic components of workflows; a workflow is a collection of programs assembled together using the other constructs.

A program instance is presented as a yellow box on the canvas, an example of which is shown in Fig. 24. The top half of the box contains the progam name (e.g. `blastall`), and the bottom half contains a number (e.g. `1`) which uniquely identifies a particular program instance; this is used to distinguish between two programs of the same name on the canvas.

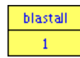

Figure 24: Job

Figure 25: Sequential dependency

Most programs have user-changeable properties. Double-clicking on the yellow box brings up the properties dialog box. This dialog box follows the user interface elements of Jemboss.

4.2.2 Dependencies (arrows)

An arrow pointing from program A to B means that program B is dependent on A, i.e. program B cannot run until A has completed running. Figure 25 shows an arrow between `formatdb:1` and `blastall:2`, i.e. to run this workflow, you must first run `formatdb` before running `blastall`.

4.2.3 Parallel container

Two vertical blue bars denote the sides of a parallel container. The workflow between these two bars are considered to be one composite component. It is a component in the sense that it is logically one unit, like a program instance, and you can draw arrows to and from other components. Figure 26 shows a parallel container in which there are two instances of `blastall`, namely `blastall:1` and `blastall:2`.

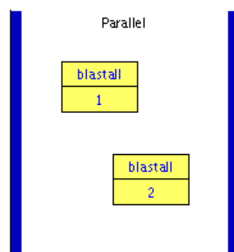

Figure 26: Parallel

4.2.4 Parallel foreach container

The *parallel for each* container is denoted by an orange rectangle with rounded corners, and a label `pforeach` with an icon in the top right corner. This container has two parameters: a variable name and a *glob pattern*. A glob pattern is used to match file names based on rules and is commonly used in many unix programs. Examples of glob patterns are `*.txt` and `seq*.fasta`. The former would match, for example, `file1.txt` and `file2.txt`, whereas the latter would match, for example, `seq002.fasta` and `seqAAyyXX82ff.fasta`.

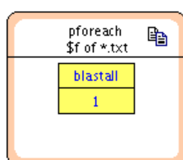

Figure 27: Parallel for each loop

During workflow execution, when its depending components, i.e. components which point to it, have completed, Wildfire will find all files which match the glob pattern, and create one independent copy of the workflow within the container for each matching file. In each copy, the value of the variable specified in the container is assigned the file name of the corresponding file. The value of this variable can be used for constructing parameters for the programs inside the container.

For example, the parallel for each container in Fig. 27 specifies variable name `$f` and glob pattern `*.txt`. Supposing files `a.txt`, `b.txt` and `c.txt` are all the files that match this pattern, then Wildfire will create three independent copies of `blastall`. The `blastall` program can refer to the value of `$f` using the `=...` notation.

4.2.5 Parallel for container

The `parallel for` container is similar to the parallel for each container. In this case, the container has a name of the variable (refered to as the `index`) and a lower and upper bound. During execution, Wildfire creates an independent copy of the workflow within for each integer between the lower and upper bounds. In each copy, the variable gives the corresponding index value.

For example, the parallel for loop in Fig. 28 specifies index variable `$f` and lower and upper indices `1` and `10`. In this case, Wildfire will create 10 copies of `blastall`, one each for `$f` in $1, 2, \ldots, 10$.

The main difference between the two types of parallel container is that the number of copies deriving from the parallel for container can be determined be-

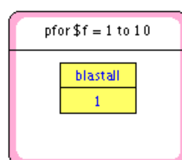

Figure 28: Parallel for loop

fore the program is run, whereas the number of copies for the parallel for each container is determined by the output of the programs themselves, and so cannot be determined until the programs have been run.

4.2.6 While loop

A purple circle is part of the *while loop*, which allows workflows to have a variable number of sequentially dependent programs. A while loop consists of a program called the `loop guard`, and a loop body. The loop body can be any workflow component.

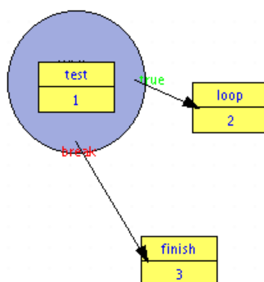

Figure 29: While loop

In the example in Fig. 29, the program `test` is the loop guard, and the body is the program `loop`. During execution, Wildfire will execute `test`, and if `test` wrote any thing to standard output, the result of execution is deemed to be *false*, otherwise it is deemed to be *true*. If `test` returns true, then Wildfire will execute the loop body, which in this example is a program called `loop`. If, however, `test` returned false, then Wildfire will follow the *break* arrow, which in this example leads to a program called `finish`. After execution of the loop body, Wildfire runs the loop guard, i.e. `test`, again, and depending on whether it returns true or false, Wildfire behaves as before.

Note that if the loop guard always returns true, then Wildfire will execute the loop body and loop guard again and again and the execution will not terminate.

It is important to note that the loop guard must be a *boolean* program. Such a program is one which does not write to any file, and signals false by writing to the standard output, and true by writing no data.

4.2.7 Conditional branches

A purple rhombus denotes a branch point. A branch point has an associated program called a *condition* and two components called the true and false branches, respectively. The condition is analogous to the loop guard of the while loop, and will return either true or false. If the condition returns true, then execution continues along the true branch; otherwise along the false branch.

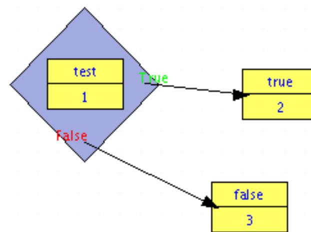

Figure 30: Conditional

The example in Fig. 30 shows a branch point with condition `test` and branches `true` and `false`.

5 Executing workflows

5.1 Export workflow as GEL script

For execution, Wildfire will export the workflow as a GEL script. The GEL script is the textual representation of the graphical canvas. This GEL script is given as input to a suitable GEL interpretor, depending on the execution platform.

The GEL script is automatically exported when the user runs the workflow. However, it is also possible to just export the workflow without executing it by select `Run -> Build`. This will save the GEL script in the project directory with the file name *`project`*`.gel` where *`project`* is the name of the project.

If desired, you can take this exported GEL script and manually invoke a GEL interpretor on the command line. This is convenient for workflows that have to be run frequently, or very long-running workflows.

The “Build” feature also serves as a useful mechanism to check for basic errors in the workflow as described in the next section.

5.2 Illegal workflows

A Wildfire workflow consists of components joined by arrows. A component can be a program instance (denoted by a yellow rectangle) or any of the pfor, pforeach or parallel containers.

Each workflow component can have at most one outbound and one inbound arrow. The arrows must satisfy two constraints: (i) they can only join components at the same level; and (ii) they must not form cycles. Figure 32 shows a workflow where the arrows do not satisfy constraint (i), and Fig. 31 shows a workflow with a cycle.

In the case of illegal workflows, Wildfire will alert the user by annotating the workflow (e.g. Figs. 32 and 31) when the user tries to run the workflow or build the GEL script.

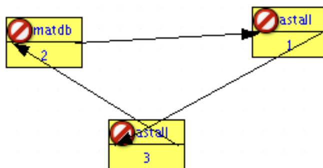

Figure 31: Cyclic workflow (error).

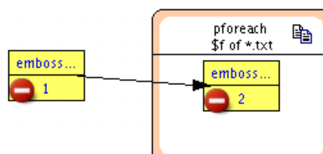

Figure 32: A workflow with an invalid dependency. The dependency arrow from embossversion:1 should connect to the pforeach box, not the embossversion:2 instance inside.

The test program is required for the if and while constructs. Figure 33 shows the canvas when the test is missing for the if construct. Similarly, Fig. 34 shows the canvas for the while construct.

5.3 GEL target platforms

Wildfire supports two modes of execution (local and remote), as well as all the various parallel platforms supported by GEL (i.e. LSF, PBS, SGE clusters, Condor Grids, and SMP servers).

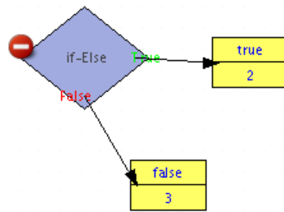

Figure 33: “If construct” with missing test.

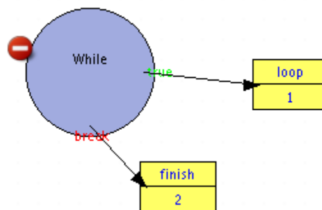

Figure 34: “While construct” with missing test.

Local execution is simpler, and assumes that the machine running Wildfire will also be running GEL. Typically, this would mean that the machine on which you are running Wildfire is in fact part of a cluster, Condor Grid or is a compute server. Since GEL presently only supports GNU-style Unix machines, this option is not available for Windows users.

Remote execution means that the project will be transfered to a remote host for execution, and the results will be transfered back. This is necessary for people who are using Windows. The remote host typically would be the head node of a cluster, any submit host of a Condor Grid, or the compute host itself. Since the project, including the data files and any binary files, must be transfered to the remote host, this mode of execution is less efficient.

For both modes of execution, the user must select which type of parallel platform the target host is. In the case of execution on a cluster, the user must select one of *Portable Batch System* (PBS), *Sun Grid Engine* (SGE) or *Load Sharing Facility* (LSF) to match the scheduler installed on the cluster. In the case of execution on a Condor Grid, the user must select Condor. In the case of a multiprocessor server, or if you do not want use any scheduler, you can simply select *Local Threads*.

To simplify these execution profiles, sets of options can be saved as profiles. When you select `Run -> Run`, Wildfire will show the execute dialog box (Fig. 35) which lists the available target platform profiles. Figure 35 shows two profiles named `local` and `remote_linux`. Profile `local` is installed by

default and will execute the workflow using the local mode, and local threads (i.e. same machine and no scheduler).

To create a new profile, click on “New”. This will bring up the dialog box displayed in Fig. 8. The fields in this dialog box are explained below.

Profile The name of this profile.

Local/Remote Local or remote mode of execution.

Hostname (Only needed for remote execution mode.) Name of remote host.

Port Port on which ssh daemon is running on the remote host. Typically this would be port 22.

Scheduler Select one of *Portable Batch System* (PBS), *Sun Grid Engine* (SGE) or *Load Sharing Facility* (LSF), or *Local Threads*.

NumProc (Only needed when Local Threads is selected.) The maximum number of concurrent processes allowed.

MPI Type / MPI Path These are reserved for future use.

Clean up Tick this box to remove the temporary directory after workflow execution. This option is only valid for the local execution mode.

Click “Save” to save the profile.

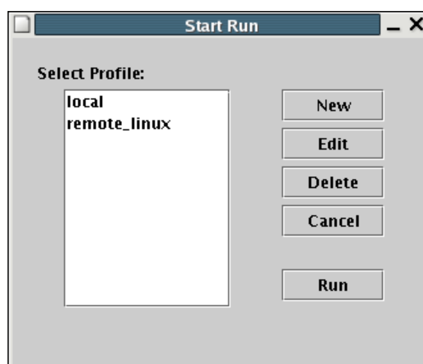

Figure 35: Wildfire’s Run dialog box. Each item in the list corresponds to a target platform profile.

6 Examples

Wildfire is distributed together with a collection of examples. If you had installed Wildfire using the Windows installer, they are available in the `C:\Documents and Settings\⟨username⟩\project\examples` directory. Otherwise, in Windows, to copy the examples, copy the directories in `c:\Program Files\Wildfire 2.0\examples` to `c:\Documents and Settings\⟨user⟩\project`.

In Linux/UNIX, copy the directories in the `examples` subdirectory of the Wildfire program directory to `$HOME/project`. For example, if you installed Wildfire in `$HOME/wildfire-2.0`, then copy the directories in `$HOME/wildfire-2.0/exampl`

Once the examples have been copied, when you select `File -> Open`, the examples will be available immediately.