



## **ECFLOW USERS GUIDE**

**For BETA version of ecFlow**

**March 2012**

This document is produced as a supplement to the online training course.

© Copyright 2012

European Centre for Medium Range Weather Forecasts  
Shinfield Park, Reading, RG2 9AX, United Kingdom

Literary and scientific copyrights belong to ECMWF and are reserved in all countries.

The information within this publication is given in good faith and considered to be true, but ECMWF accepts no liability for error, omission and for loss or damage arising from its use.

## Table of Contents

TABLE OF CONTENTS .....	1
<b>1 INTRODUCTION TO ECFLOW.....</b>	<b>4</b>
1.1 WHAT IS ECFLOW?.....	4
1.2 HISTORY .....	4
1.3 ECFLOW EXECUTABLES .....	5
1.4 TERMINOLOGY AND TYPOGRAPHY .....	5
<b>2 RUNNING ECFLOW .....</b>	<b>6</b>
2.1 OVERVIEW .....	6
2.2 STATUS OF A TASK.....	7
2.3 STATUS OF A FAMILY OR SUITE.....	8
2.4 USING ECFLOW.....	9
2.5 SUITE DEFINITIONS .....	9
2.6 WRITING ECFLOW SCRIPTS .....	10
2.7 TIME CRITICAL TASKS.....	11
2.8 RUNNING TASKS ON REMOTE SYSTEMS.....	12
2.9 MOVING SUITES BETWEEN ECFLOW SERVERS .....	13
2.10 DEBUGGING DEFINITION FILES .....	14
2.11 DEBUGGING ECFLOW SCRIPTS.....	14
<b>3 THE ECFLOW PRE-PROCESSOR .....</b>	<b>16</b>
3.1 LOCATING '.ECF' FILE.....	17
3.2 ECFLOW MANUAL PAGES.....	18
3.3 INCLUDE FILES.....	20
3.4 COMMENTS.....	22
3.5 STOPPING PRE-PROCESSING .....	22
3.6 ECF_MICRO.....	23
3.7 HOW A JOB FILE IS CREATED FROM AN ECFLOW FILE.....	24
<b>4 ECFLOW VARIABLES .....</b>	<b>25</b>

4.1	VARIABLE INHERITANCE .....	25
4.2	ECFLOW SERVER ENVIRONMENT VARIABLES.....	26
4.3	ENVIRONMENT VARIABLES FOR THE ECFLOW CLIENT.....	27
4.4	ECFLOW SUITE DEFINITION VARIABLES .....	28
4.5	GENERATED VARIABLES .....	30
4.6	VARIABLES AND SUBSTITUTION.....	33
<b>5</b>	<b>TEXT BASED SUITE DEFINITION FORMAT.....</b>	<b>35</b>
5.1	DEFINING A SUITE, USING THE TEXT DEFINITION FILE FORMAT .....	36
5.2	DEPENDENCIES.....	44
5.3	ATTRIBUTES.....	53
<b>6</b>	<b>DEFINING A SUITE USING THE PYTHON API.....</b>	<b>60</b>
6.1	PYTHONPATH AND LD_LIBRARY_PATH .....	60
6.2	DEFINING A SUITE, USING THE PYTHON API .....	60
6.3	DEPENDENCIES.....	61
6.4	ATTRIBUTES.....	63
6.5	CONTROL STRUCTURES AND LOOPING .....	64
6.6	ADDING EXTERNS AUTOMATICALLY.....	64
6.7	CHECKING THE SUITE DEFINITION .....	65
6.8	CHECKING JOB GENERATION.....	65
6.9	HANDLING DUMMY TASKS .....	66
6.10	SIMULATION OF A RUNNING SUITE .....	67
6.11	ERROR HANDLING .....	68
<b>7</b>	<b>THE ECFLOW SERVER.....</b>	<b>69</b>
7.1	STARTING THE ECFLOW SERVER.....	69
7.2	STOPPING ECFLOW SERVERS .....	70
7.3	CHECKING IF AN ECFLOW SERVER IS RUNNING ON A HOST.....	70
7.4	START-UP FILES FOR ECFLOW SERVER .....	71
7.5	SECURITY .....	73
7.6	SECURITY: ECFLOW WHITE LIST FILE.....	73

7.7	HANDLING OUTPUT .....	74
<b>8</b>	<b>ECFLOW CLI (COMMAND LEVEL INTERFACE) .....</b>	<b>76</b>
8.1	GET.....	76
8.2	CLI SCRIPTING IN BATCH.....	76
8.3	CONFIGURING ECFLOW .....	77
8.4	COMPILER AND OS REQUIREMENTS.....	78
<b>9</b>	<b>FLAGS USED BY ECFLOW.....</b>	<b>79</b>
<b>10</b>	<b>ECFLOWVIEW .....</b>	<b>81</b>
10.1	MAIN WINDOW MENUS.....	82
10.2	ECFLOWVIEW "BUTTONS" .....	86
10.3	MENUS .....	88
10.4	THE COLLECTOR .....	88
10.5	SEARCHING .....	89
10.6	DEPENDENCIES.....	90
10.7	EDITING SCRIPTS.....	90
10.8	ZOMBIES.....	91
<b>11</b>	<b>INDEX .....</b>	<b>93</b>

# 1 Introduction to ECFLOW

## 1.1 What is ECFLOW?

ECFLOW is a client/server workflow package that enables users to run a large number of programs (with dependencies on each other and on time) in a controlled environment. It provides reasonable tolerance for hardware and software failures, combined with restart capabilities.

ECFLOW submits *tasks* (jobs) and receives acknowledgments from the tasks when they change *status* and when they send *events*, using *child* commands embedded in your scripts. ECFLOW stores the relationships between tasks, and is able to submit tasks dependant on triggers, such as when a given task changes its status, for example when it finishes. Users communicate with ECFLOW server using:

- Command Level Interface (CLI).
- Python interface
- *ecflowview*. This is an X-Windows/Motif based program.

ECFLOW runs as a server receiving requests from clients. CLI, *ecflowview* and the suite jobs are the clients. Communication is based on TCP/IP. Note that ECFLOW is a scheduler and is not a queuing system such as NQS, SGE, Load leveller or PBS. However, it can submit to queuing systems.

## 1.2 History

ECFLOW is a product of the European Centre for Medium-range Weather Forecasts (ECMWF) and was written by the Meteorological Applications (MetApps) section.

ECFLOW is a complete replacement of SMS. It has been written using an object oriented language. It provides most of the functionality that is available in SMS.

### 1.2.1 Difference with SMS

- Maintenance and enhancement of server software easier.
- SMS provided a custom scripting language for defining suites. In ECFLOW you have the choice of using either a simple text based format or a Python API as a replacement for CDP/text interface. The entire suite definition structure can be specified in python.
- Any language can be used to create the textual suite definition file as it has a published format. This is different to current SMS where suite definition keywords are treated as commands.
- Allows better error checking and handling of zombies.

- Will work on 64 bit operating systems like AIX without the need to compile in 32 bit mode.
- Does not require an explicit login.

### 1.3 ECFLOW executables

ECFLOW functionality is provided by the following executables and shared libraries

- ecflow\_client: This executable is a command line program; it is used for all communication with the server. This executable implements the Command Level Interface (CLI). The bulk of this functionality is also provided by the python API
- ecflowview: This is a specialised GUI client that monitors and visualises the node tree hierarchy.
- ecflow\_server: This executable is the server. It is responsible for scheduling the jobs and responding to ecflow\_client requests.
- ecflow.so , libboost\_python.so: These shared libraries provide the python API for creating the suite definition and communication with the server.

### 1.4 Terminology and typography

Command names in this text are written in **this fashion**, and corresponding sections in the manual pages is shown within parentheses. UNIX manual page sections are numbered from one to eight; for example, the UNIX C shell is shown as **ksh(1)** which indicates it is in the first section of the UNIX manual pages. **load (CLI)** indicates the command **load** is a CLI command(i.e. implemented by the ecflow\_client executable)

Through the rest of this document, 'ECF' will be used interchangeably with ECFLOW. Any references to server, client or GUI, refers to ECFLOW server, client and view.

A **concept** is a name used in this document to describe an idea.

A **variable** is used a lot throughout this document.

A **node** represents a task, family, suite or equivalent when it does not matter which we are talking about.

Extracts from script or definition files look like this

```
# This is in a file
and this is important in this file
some other lines
```

Colours are also used to highlight the status of a node, so if node **/x/y/z** is active, it would look like **/x/y/z**.

## 2 Running ECFLOW

### 2.1 Overview

To use ECFLOW you need to carry out a few steps

- Write a *Suite definition*. This shows how your various tasks run and interact. *Tasks* are placed into *families* which themselves may be placed into families and/or *suites*. All these entities are called *nodes*
- Write your scripts (.ecf files); these will correspond with Task's in the *suite definition*. The script defines the main work that is to be carried out. The script includes *child commands* and special comments and manual sections that provide information for operators.

The *child commands* are a restricted set of client commands that communicate with the server. They inform the server when the job has started, completed or set an *event*.

Once these activities are done, the ECFLOW server is started and the suite definition is loaded into the server.

- The user then initiates *task scheduling* in the server
- *Task scheduling* will check dependencies in the suite definition every minute. If these dependencies are free, the server will submit the task. This process is called *job creation*.

The process of *job creation* includes:

- Locating '.ecf' script, corresponding to the Task in the *suite definition*
- Pre-processing the '.ecf' file. This involves expanding any includes files, removing comments and manual sections, and performing *variable substitution*.
- The steps above transforms an '.ecf' script to a job file that can be submitted.

The running jobs will communicate back to the server using *child commands*. This causes *status changes* on the *nodes* in the server and flags can be set to indicate various events.

ECFLOW has a specialised GUI client, called ecfview. This is used to visualise and monitor:

- The hierarchical structure of the *suite definition* (Suite, Family, Task's i.e. *nodes*)
- *state changes* in the *nodes* and the server
- Attributes of the *nodes* and any *dependencies*
- Script file '.ecf' and the expanded job file

In addition it provides a rich set of client commands that can interact with the server.

The following sections will provide more detail on the overall process.

## 2.2 Status of a task

Each task in ECFLOW has a *status*. Status reflects the *state* the node is in. In ecfview the background colour of the text reflects the status.

The status of a task can vary as follows (default colours are shown):

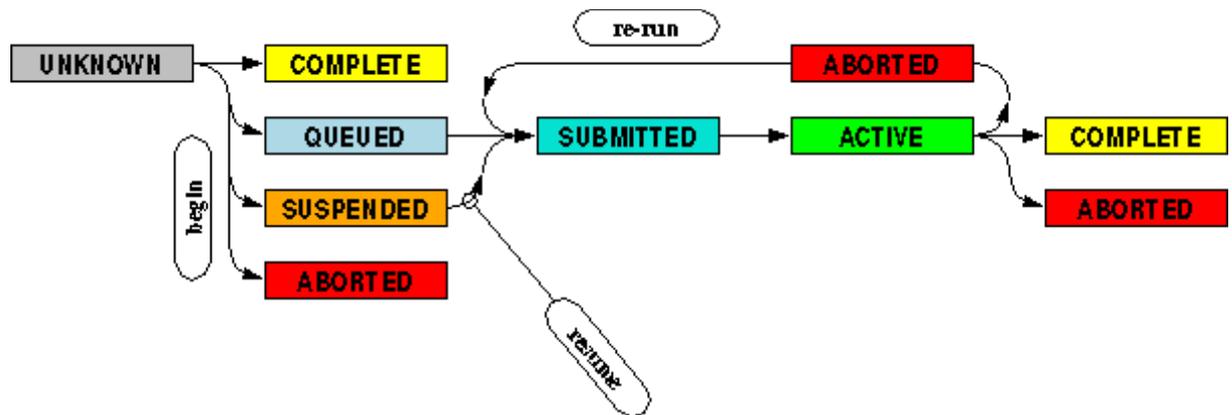
- After the **load(CLI)** command the tasks status is **unknown**
- After **begin (CLI)** command the tasks are either **queued**, **complete**, **aborted** or **suspended**. A suspended task means that the task is really queued but it must be *resumed* by the user first before it can be submitted.
- Once the *dependencies* are resolved a task is *submitted* by ECFLOW. The status is **submitted** or **aborted** (aborted, if scripts not found, variable substitution fails, or **ECF\_JOB\_CMD** failed.)
- If the submission was successful, then once the job starts, it should become **active**, this achieved when the job calls: **ecflow\_client --init**
- Before a job *ends* it may send other messages to ECFLOW such as those shown below; These are referred to as *child commands*
  - **ecflow\_client --event** Set an *event*
  - **ecflow\_client --meter** Change a *meter*
  - **ecflow\_client --label** Change a *label*
  - **ecflow\_client --wait** Wait for an expression to evaluate
  - **ecflow\_client --abort** Send an abort message to the task
  - **ecflow\_client --msg** Write a message line into ecFlow logfile
  - **ecflow\_client --status** checks task's status
- Jobs end by becoming either **complete** or **aborted** by the job itself sending a message back to ECFLOW server. The complete is set by calling: **ecflow\_client --complete** in the job file

At any time the user can *suspend* a task. The tasks status is *saved*.

The above colours are the default, but they can be changed in ecfview using the menu edit: preferences followed by the colours tab.

Figure 2-1 shows the normal status changes for a task with default colours:

Figure 2-1 Status changes for a task



### 2.3 Status of a family or suite

The status of a family or suite is the inherited most significant status of all its children.

Table 2-1 Table of importance of a nodes status shows the order of importance of the different statuses and some examples of the result of a status of the family depending on its children.

Table 2-1 Table of importance of a nodes status

Unknown	Least significant
Complete	
Queued	
Submitted	
Active	
Suspended	Will hide the underlying status
Aborted	Most important for a task, family or a suite
Shutdown	Only for Server
Halted	Most important, Only for Server

Table 2-2 Example of how the status of a family is reported.

Status of a family											
After begin command		First job sent		Second job sent		A few tasks running		One task aborts!		In the end, complete	
family	task1	family	task1	family	task1	family	task1	family	task1	family	task1
	task2		task2		task2		task2		task2		task2
	task3		task3		task3		task3		task3		task3
	task4		task4		task4		task4		task4		task4

The status of the ECFLOW server itself can be:

- **Shutdown**, Server is not scheduling jobs anymore, but allows tasks to communicate.
- **Halted**, Server is not scheduling and does not allow tasks to communicate with it. This is the default status when server is started and is needed, since **recovery** is only possible if the server is halted.

## 2.4 Using ECFLOW

To use ECFLOW you need to carry out a few steps. Firstly you need an ECFLOW server running (See Section 0). Next you need to write a suite definition that shows how your various tasks run and interact. You also need to write your ecFlow tasks/scripts. Finally your suite definition is “loaded” onto an ECFLOW server and your suite is started as required and monitored via ecfview. The process is described with the associated ECFLOW online course.

## 2.5 Suite Definitions

Once you have the server running you can **define** a suite to run on it. The suite is described by a **suite definition file**. This is covered more completely in sections 5 and 6.

A suite definition in text format will have a structure similar to the following:

```

1 # Definition of the suite test
2 suite test
3     edit ECF_HOME /tmp/COURSEDIR
4     task t1
5 endsuite

```

1. The first line is a comment line. Any characters between the # and the end of line are ignored.

2. Defines a new suite by the name of test. Only one suite can be defined in a definition file. Though a suite can contains details of more than one suite.
3. Defines the ECFLOW variable `ECF_HOME`. This variable defines the directory where all the UNIX files that will be used by the suite test will reside.
4. Defines a task named `t1`.
5. The last line finishes the definition of the suite test

Defining suites using the Python API is discussed in section 6.

## 2.6 Writing ECFLOW scripts

The *ECFLOW script* describing the **task t1**, as defined in the previous section, refers to an `.ecf` file. This is similar to a UNIX shell script. The differences, however, includes the addition of “C” like pre-processing directives and ECFLOW variables.

By default the ECFLOW pre-processing directives are specified using the `%` character. A simple example task file is given below.

```
%include <head.h>
echo "I am testing an ECFLOW script in %ECF_HOME%"
%include <tail.h>
```

Before submitting the task, ECFLOW will parse the script for ECFLOW directives and substitute relevant strings.

In the example above the `%include` command will be substituted with the content of the file `head.h` and `%ECF_HOME%` will be substituted with the ECFLOW variable `ECF_HOME`

ECFLOW scripts communicate with ECFLOW server via child commands, the `head.h` file can be used to send relevant child commands to inform `ecFlow` of the job status, set up error trapping (`ecflow_client --abort`) and define variables relating to the job environment. The `tail.h` file can contain related child commands (`ecflow_client --complete`) and information on how to clean up after the task.

Operationally at ECMWF we also include a number of header files that also setup relevant information for job scheduling via external queuing systems (such as `loadleveler` on IBM systems) and suite configuration.

### 2.6.1 Guidelines when writing operational scripts

The following are a few guidelines we use when writing operational scripts:

- All operational tasks should be rerunnable, or at the very least include instructions in the *manual page* how to restart the task (e.g. by running another task first).
- Tasks should be able to run independently of the server (again the manual page should have instructions how to restart on a different server).

- The critical parts of a suite should be independent as far as possible from the less critical parts. For instance ECMWF keeps its operational archiving tasks in a separate family from the time critical tasks.
- You should be consistent in your use of scripts.
- You should turn on error failing so ECFLOW can trap failures, e.g. using set -e and trap in ksh.
- You should not use UNIX aliases.
- You should not use shell functions as these can cause problems trapping errors.(or explicitly repeat the trapping to ensure portability)
- Exported variables should be UPPERCASE.
- Defined ECFLOW variables should not start with generated ECF\_ to avoid confusion.
- All variables should be set (using default values if necessary).
- Try to avoid using NFS mounted file systems in the critical path.
- Use files or file-systems owned by one single “operational” user
- Always clean up. i.e. Job output, server logs, job scripts.
- Tasks should run in a reasonable time.
- Keep the ECFLOW scripts manual page up to date, including details of how to handle failures and who is responsible for the script.
- Keep your task output - at the end of each day we keep the output of all tasks, tar them up and store them on tape. This allows us to review suites at a later date and is useful in indicating when problems may have started.
- Avoid duplication of scripts. Scripts can easily be made configurable and shareable.

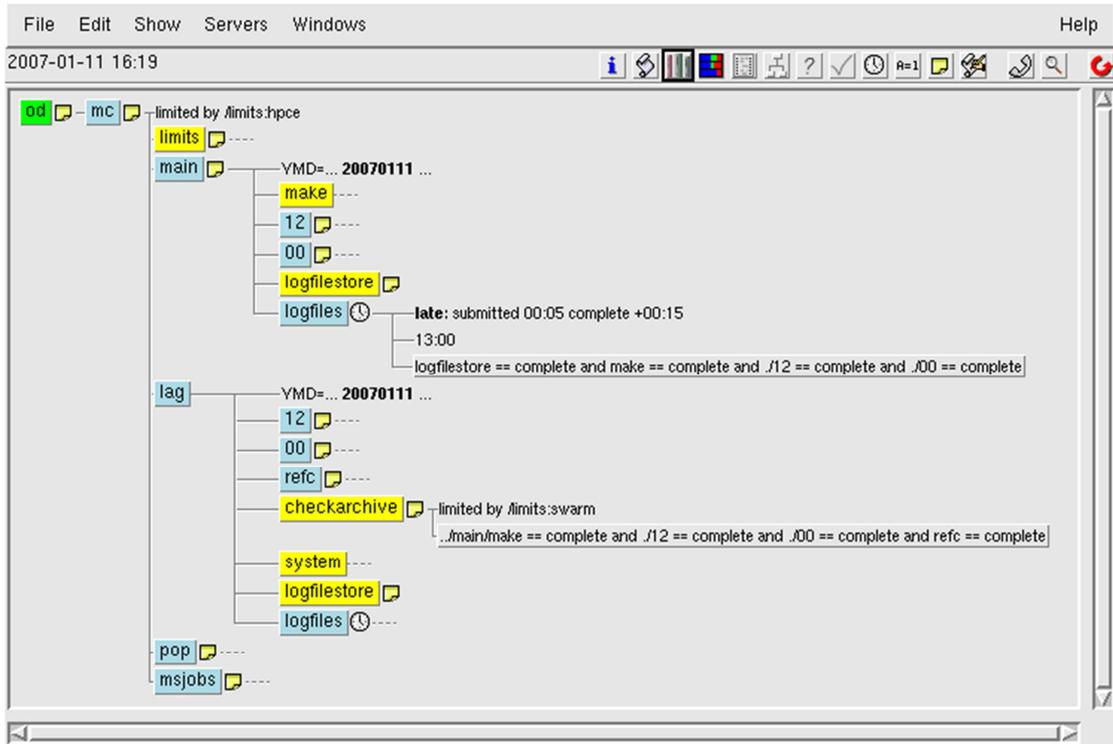
## 2.7 Time Critical tasks

At ECMWF we use ECFLOW to schedule our operational suites. We also separate out critical and non-critical tasks, thus allowing our operators to more easily monitor suites. In the Figure 2-2 we show the coarse structure of one of our operational suites.

The suite is divided into four sections: “main” handles the time critical parts of the suite, such as the actual model, “lag” handles the archiving and other non time-critical tasks, “pop” handles the plotting of results and “msjobs” handle the submission of member state jobs.

Note that each family has its own date *repeat* labelled YMD. This allows us to use triggers including the suite date (YMD) and also allows the less critical tasks to run even a few days behind if necessary. This is useful when running a test suite not in real-time.

Figure 2-2 Sample suite structure



ECFLOW can help in the monitoring of suites in many ways, beyond the indication of task status. For instance the **late** command in ECFLOW can be used to highlight problems with time critical scripts. The command will mark a node as late when certain conditions are met; such as submitted for too long, running for too long or not active by a certain time. This is used in conjunction with `ecflowview` to launch a pop up window once a late condition is reached. To use this option you need to make sure that the `ecflowview` option “show/special/late nodes” is selected.

In a number of our suites we have also defined check tasks that interrogate ECFLOW using the `status` command to find out if tasks have, for instance, completed at a given time.

## 2.8 Running tasks on remote systems

To start a job, ECFLOW uses the content of the `ECF_JOB_CMD` variable. By modifying this variable, it is possible to control where and how the job will run. The command should use the ECFLOW variables `ECF_JOB` and `ECF_JOBOUT`. `ECF_JOB` contains the name of the job file and `ECF_JOBOUT` contains the name of the file that should contain the output. For a `ksh` or `bash` UNIX script the default command is:

```
%ECF_JOB_CMD% 1> %ECF_JOBOUT% 2>&1 &
```

To run the tasks on a remote machine you can use the UNIX command `rsh` (or `ssh`). We would like the name of the host to be defined by an ECFLOW variable called `HOST`. We

assume that all the files are visible on all the hosts, using NFS. You can then redefine the ECF\_JOB\_CMD as follows for ksh:

```
edit ECF_JOB_CMD "rsh %HOST% '%ECF_JOB% > %ECF_JOBOUT% 2>&1 &'"
```

As ecFlow makes use of standard UNIX permissions you may experience problems using `rsh`. Make sure that the file `$HOME/.rhosts` contains a line with your username and the machine where your ECFLOW is running.

If your login shell is `csh`, you can define `ECF_JOB_CMD` as:

```
edit ECF_JOB_CMD "rsh %HOST% '%ECF_JOB% >& %ECF_JOBOUT%'"
```

You can also submit tasks directly to the relevant queuing system on the target machine. In fact at ECMWF, we have written a UNIX script to submit tasks to multiple systems and multiple queuing systems (`ecf_submit`). An example `ecf_submit` script is included with the ecFlow release.

```
edit ECF_JOB_CMD "ecf_submit %USER% %SHOST% %ECF_JOB%  
%ECF_JOBOUT%"
```

Alongside this we include into our 'ecf' scripts a generic script header containing typical queuing commands (such as wall clock time and priority), e.g.

Contents of sample `qsub.h`

```
# QSUB -q %QUEUE%  
# QSUB -u %USER%  
# QSUB -s /bin/ksh  
# QSUB -r %TASK%_%FAMILY1:NOT_DEF%  
# QSUB -o %LOGDIR%%ECF_NAME%.%ECF_TRYNO%  
# QSUB -lh %THREADS:1%
```

The `ecf_submit` script can replace these generic queuing commands with the relevant commands for the host to which the task is submitted and submit the task relevant way, e.g. for a PBS system it replaces the QSUB commands with the equivalent PBS commands.

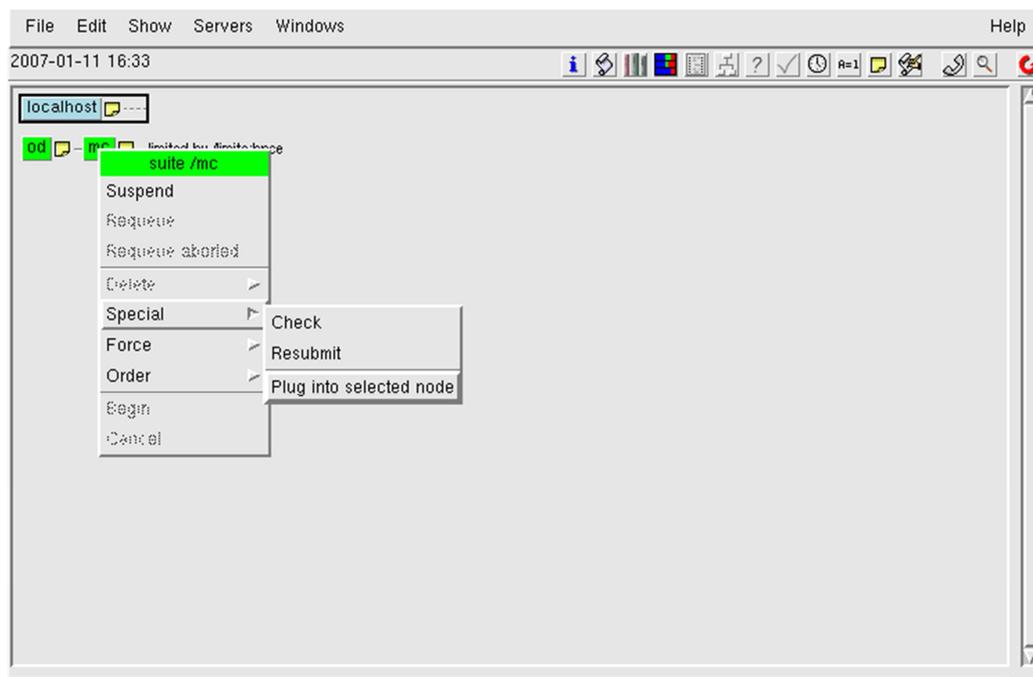
Similarly to running a task remotely, to kill a task remotely you need to either send a signal 2 (kill -2) to the task or issue the relevant queuing system command. Again we have included all this information into a script called "ecf\_kill" that issue the correct command depending on the host. This and other example scripts "ecf\_status" (show status of tasks) and "ecfurl" (open a web link for a task) are included in the latest releases of ECFLOW.

## 2.9 Moving suites between ECFLOW servers

When testing a suite, you may want to initially run on a test server. Once operational you may then wish to move the suite onto another server. Whilst you can replay the suite on the new server you can also use the CLI `swap` command to move the suite. However, a more simple method is to use `ecflowview` to *plug* the suite into a new server.

To do this in ecfview you left click on the destination server. Then right click the suite you want to move and select the “special/plug into selected node” option. ecfview will then move your suite (see Figure 2-3).

**Figure 2-3 - Plugging a suite between ECFLOW servers**



## 2.10 Debugging definition files

You can test a definition file using the command:

```
ecflow_client --load=/my/home/exotic.def check_only
```

This will check that the suite definition is correct and can be loaded into the server.

## 2.11 Debugging ECFLOW scripts

When debugging scripts you need to consider the process used to submit the job. First a script is accessed and expanded to create the job file. This job file is then submitted to the relevant system where it is run, possibly via a queuing system. Errors can occur at any of these steps.

### 2.11.1 Location of the ECFLOW script

The first thing to check is whether the job file can be created. In ecfview you should be able to view your script. If not, you should receive a pop up window indicating a file read error. This indicates that ECFLOW cannot find your script as either it does not exist or ecFlow cannot find it. Look at the value of the `ECF_SCRIPT` variable to see where ecFlow expects to find the file. The location can be modified using the variable `ECF_FILES` as described later.

### **2.11.2 Creation of the ECFLOW Job File**

The next thing ecFlow will do is create the job file by adding all include files and substituting ecFlow variables (see the next chapter on “The ECFLOW Pre-processor” for more details). To test if ecFlow can find all the include files and variable click the edit button in ecflowview. If you get the error message “send failed for the node” ecFlow may not be able to access the include files or some ecFlow variables have not been set. More details will be given in the ecFlow log files which can also be seen in the “history” when right clicking on an ecFlow server in ecflowview.

### **2.11.3 Submission of the ECFLOW job file**

The next stage is to submit the ecFlow job file. The best way to debug this is to try the submission of the job file on the command line as described by your ECF\_JOB\_CMD variable. This will usually show up any problems in the job submission process. The script we use to submit our job files also makes visible the job submission output in the ECF\_JOBOUT directory.

### 3 The ECFLOW Pre-processor

The pre-processor in ecFlow reads the ecFlow files and processes them to form a job-file, manual-page or for editing in ecflowview. It also does the interpretation of the commands defined in `ECF_JOB_CMD` and `ECF_KILL_CMD`.

The *ecFlow pre-processor* was developed to be part of the ECF. It allows users to do some of the *C-pre-processor* like tasks, namely *including* files. It works on the same principle by looking at the first character on each line in *ecFlow file* (or *ecFlow script*). If that is found to be an *ECF-micro character* (by default '%', see `ECF_MICRO`) the line is for the pre-processor. If, however, the line starts with two of these characters, a single '%' character is passed on to the next phase (to be used as an ECF-variable introducer).

Currently there is no %if - statement in the pre-processor. If - statements, however, can be handled by the shell running the script.

The pre-processor also carries out variable *substitution*. When ecFlow is preparing to execute a task it reads the ecFlow script and produces a job file in which it replaces all the relevant variables.

Table 3-1 shows pre-processor symbols that ecFlow understands. Notice that some of them work in pairs.

Table 3-1 ECFLOW pre-processor symbols

<b>%include &lt;filename&gt;</b>	<code>%ECF_INCLUDE%/filename</code> directory is searched for the <b>filename</b> and the contents included into the output. If that variable is not defined <code>ECF_HOME</code> is used instead. This is the recommended format for include.
<b>%include "filename"</b>	Include the contents of file <code>%ECF_HOME%/SUITE%/FAMILY%/filename</code> into the output
<b>%include filename</b>	Include the contents of file <b>filename</b> into the output. Notice that since the <code>\$CWD</code> of ecFlow can be anywhere, the only form that can be safely used must start with slash '/'.
<b>%includenopp filename</b>	Same as <b>%include</b> , but the file is not interpreted at all. This allows you to test the filename separately with ease. (Same three formats for <i>filename</i> as for plain <b>%include</b> .)
<b>%comment</b>	Remove all the lines from the output until a line with <b>%end</b> is

	found.
<b>%manual</b>	If creating a <i>job-file</i> remove all the lines from the output until a line with <b>%end</b> is found. If creating a <i>manual page</i> include all the lines until a line with <b>%end</b> is found.
<b>%nopp</b>	Stop the pre-processing until a line starting with <b>%end</b> is found. No interpretation of the text will be done (e.g. no variable substitutions)  Line is retained, if pre-processing is requested by ecflowview
<b>%end</b>	End processing of <b>%comment</b> or <b>%manual</b> or <b>%nopp</b>
<b>%ecfmicro CHAR</b>	Change the ECF_MICRO character to the character given. If set in an include file the effect is retained for the rest of the job (or until set again).  This does not change how <b>ECF_FETCH</b> or <b>ECF_JOB_CMD</b> work, they still use <b>ECF_MICRO</b>

Note that for **%include**, if the **filename** starts with slash, '/' character, no interpretation will be made. The full path name of the file will be used.

Currently you cannot have variables in the include file definition.

```
%include <%SUITE%/file.h> # is NOT ALLOWED!!!
```

### 3.1 Locating '.ecf' file

ecFlow looks for files using the following search process, when trying to locate the '.ecf' associated with a task.

- a) First it uses the variable **ECF\_SCRIPT** and tries to open that file. **ECF\_SCRIPT** is generated from **ECF\_HOME/SUITE/FAMILY/TASK** and the try number, which is available as variable **ECF\_TRYNO**.
- b) Otherwise if variable **ECF\_FILES** exists, it must point to a directory which is searched in reverse order, e.g. let's assume that the node name is **/o/12/fc/model** and that **ECF\_FILES** is defined as **/home/ecmwf/emos\_ECF/def/o/ECFfiles**. The order of files tried is as follows:
  1. **/home/ecmwf/emos\_ECF/def/o/ECFfiles/o/12/fc/model.ecf**
  2. **/home/ecmwf/emos\_ECF/def/o/ECFfiles/12/fc/model.ecf**
  3. **/home/ecmwf/emos\_ECF/def/o/ECFfiles/fc/model.ecf**

4. /home/ecmwf/emos\_ECF/def/o/ECFfiles/model.ecf

This may at first may be seen as over kill, but you can put all the files for a number of suites in one distinct file system/directory.

If the original `ECF_SCRIPT` did not exist, ecFlow will check the directories for the job file in `ECF_HOME` (`ECF_SCRIPT` is derived from `ECF_HOME`). If a directory does not exist, ecFlow will create it. This helps to clean up old job-files and output and makes the maintenance of the scripts easier. It also guarantees that the output can be redirected into the file without the job creating the directory. (e.g. NQS option `QSUB -ro`, or when using redirection.)

Using `ECF_FILES` means that you do not have to create and maintain a link-jungle, e.g. the `model.ecf` above exists in a number of different families in ECMWF operational suites. The file is placed in a directory `.../ECFfiles/fc/` and used by nodes `/o/00/fc/model`, `/o/12/fc/model` etc. This *trick* works nicely as long as there are no other tasks named `model` in the same family.

## 3.2 ECFLOW Manual pages

*Manual pages* are part of the ecFlow script. This is to ensure that the manual page is updated when the script is updated. The manual page is a very important operational tool allowing you to view a description of a task, its importance, task dependencies and possibly describing solutions to common problems. The ecFlow pre-processor can be used to *extract* the manual page from the script file to be viewed by ecflowview. The manual page is the text contained within the `%manual` and `%end` tags. They can be seen using the manual button on ecflowview.

Manual pages are a vital source of information to users. The text on manual pages is not copied into the job-file when ecFlow sends a task into execution. Suites, families and tasks can have manual pages. Manual pages for tasks are placed in the ecFlow script inside a pair of *pre-processor lines* as in the following example:

```
%manual
  OPERATORS: If this task fails, set it complete and report
              next working day
  ANALYST:   Check something or do something clever!
%end
ls -l
pwd
hostname
%manual
  Rest of the manual page is placed here, closer to the code
%end
```

There can be multiple manual sections in the same file. When viewed they are simply concatenated. This helps in maintaining the manual pages. It is good practice to modify the manual pages when the script is changed.

Viewing manual page from the above ecFlow script would look something like

```
OPERATORS: If this task fails, set it complete and report
            next working day

ANALYST:    Check something or do something clever!

Rest of the manual page is placed here, closer to the code
```

After `%manual` all pre-processor symbols are ignored until `%end` is found. Thus you cannot use `%comment` - `%end` to *un-comment* manual pages. Manual pages may have *include statements* like in the following extract:

```
%manual
OPERATORS: If this task fails, set it complete and report
            and next working day

ANALYST:    Check something or do something clever!

%include <manual/foo.bar>
%end

ls -l
pwd
hostname

%manual
Rest of the manual page is placed here
%end
```

For example standard instructions for operators could be placed in a single file and then included in every task (like contact phone numbers etc.) How the include file is found is explained in the next section.

Suites and families can also have manual pages. However, these are separate files placed in the suite/family directories e.g. the manual page for a **family family1** is a file `family1.man` in the relevant directory.

### 3.3 Include files

*Include files* are used where same piece of code can be inserted into multiple files. This allows all files using that include file to be easily changed. A group of files may have their own include file; e.g. all the tasks in an archiving family, could include one common file for the variable definitions needed. This makes the maintenance of the tasks much easier.

In the same way as the *C-pre-processor*, ecFlow *include files* do *nest*. There is no limit within ecFlow on how many times they nest beyond system limitations.

In the simplest case an ecFlow file would have at least two *include statements*. One include at the beginning and one at the end of the file. An example is given below. There are two extra lines apart from the lines needed for the task itself. This helps to understand the script since only lines needed for this task are visible. The extra *ecFlow code* is not visible.

```
# Example of using include statements in ecFlow file
#include <head.h>

# do the steps for the task

#include <end.h>
```

When ecFlow needs to read an include-file it tries to locate them from the directory pointed to by variable **ECF\_INCLUDE** (unless full path name was given.) Typically this variable is set in the suite definition file at the same time as **ECF\_FILES**.

The start of the definition for a suite will normally be something like:

```
suite my_suite
  edit ECF_FILES    /home/ma/map/def/$SUITE/ECFfiles
  edit ECF_INCLUDE /home/ma/map/def/$SUITE/include
  edit ECF_HOME    /tmp/map/ECF
  ...
```

You need to declare the ECF-variables needed. In the start of an ecFlow script you need to make sure that any command failing will be trapped and calls

```
#> ecflow_client --abort="<Reason>"
```

You also need to tell ecFlow that the task is active by using

```
#> ecflow_client - -init <process id>
```

In a large suite, with hundreds of tasks, you would need to execute the same commands in each of them. Editing just a single (header) file is somewhat easier than editing them all.

E.g. **file head.h**

```
#!/bin/ksh
ECF_NAME=%ECF_NAME%
ECF_NODE=%ECF_NODE%
ECF_PASS=%ECF_PASS%
ECF_PORT=%ECF_PORT%
ECF_TRYNO=%ECF_TRYNO%
ECF_RID=$$
export ECF_NAME ECF_NODE ECF_PASS ECF_TRYNO ECF_PORT ECF_RID
ERROR() { echo ERROR ; ecflow_client --abort=trap; exit 1 ; }
trap ERROR 0
trap '{ echo "Killed by a signal"; ERROR ; }' 1 2 3 4 5 6 7 8 10 12
13 15      # list using kill -l or man kill
set -e
ecflow_client --init=$$
```

The same applies to the end of the task. You want to tell the ecFlow that the task is complete by using **ecflow\_client --complete(CLI)** and **un-trap** the shell.

e.g. **file tail.h**

```
ecflow_client --complete
trap 0
exit
```

Generally you would have more than just a single **include file** at the beginning of an ecFlow file, e.g. one to have common options for your queuing system, then a few lines for the queuing options unique to that job. There may be an include file to specify options for an experimental suite, and so on. There are around ten different include files used in the ECMWF operational suite.

## 3.4 Comments

Comments are enclosed between pre-processor lines **%comment** and **%end**. These lines are then ignored when the file is being pre-processed, either for running the job or for the manual-page. You cannot, however, *comment-out* a part of the manual page.

Since comments are processed before a job is created they can be placed anywhere in the script. Just remember that comments do not *nest* and that you cannot use comments inside manual pages. (You can of course change the **%manual** to be **%comment**)

The following extract is an example of using comments:

```
mars << EOF
  RETRIEVE ,
  PARAM=10U/10V,DATE=... ,
  %comment
  temp mod by OP / 1.1.2012 BC
  TARGET= "/xx/yy/zz" ,
  %end
  TARGET= "zz" ,
  END
EOF
```

And the following is the corresponding output when pre-processed.

```
mars << EOF
  RETRIEVE ,
  PARAM=10U/10V,DATE=... ,
  TARGET= "zz" ,
  END
EOF
```

## 3.5 Stopping pre-processing

The ecFlow pre-processor allows parts of the ecFlow script to be included *as is* or without being pre-processed. This was done mainly to make it easy to use languages such as **perl** which make significant use of the %-sign. Pre-processing can be stopped in two ways

By using a pair of lines: **%nopp** and **%end** which will completely stop the pre-processing between those lines

By using `%nopp filename` which will include the file as is without any interpretation. This makes it easy to test the script separately, but allows it to be edited by `ecflowview`.

```
%nopp
echo "char like % can be safely used here"
date +%Y.%m.%d
%end

echo "otherwise we must write"
date +%Y.%m.%d
```

### 3.6 ECF\_MICRO

The variable `ECF_MICRO` can be set to change the ecFlow micro-character. This affects both the commands and the interpretation of the ecFlow files. The default value is "%".

An example showing changing the micro-character follows:

```
suite x
  edit ECF_MICRO "&"
  family f
    task t
  File t.ECF
  &include <head.h>
  echo job here
  &include <end.h>
```

Another way of changing the micro-character is to set it up in the ecFlow script. It only effects the script interpretation not the commands `ECF_JOB_CMD` or `ECF_KILL_CMD`.

### 3.7 How a job file is created from an ecFlow file

The job file is the actual file that ecFlow will submit to the system. Starting with the following ecFlow file:

**task.ecf**

```
%manual
  OPERATORS: Set the task complete and report next day
%end
%include <head.h>

echo do some work
sleep %SLEEPTIME%
echo end of job

%include <end.h>
```

This uses the header files **head.h**, **end.h** for example as given earlier and with SLEEPTIME defined as having a value **60**.

After pre-processing the job-file will include the header files and variables and exclude comments and man pages. It would look something like:

**task.job1**

```
#!/bin/ksh

ECF_NAME=/suite/family/task
ECF_NODE=localhost
ECF_PASS=xYz12AbC
ECF_PORT=3141
ECF_TRYNO=1

export ECF_NAME ECF_NODE ECF_PASS ECF_TRYNO ECF_PORT
ERROR() { echo ERROR ; ecflow_client --abort=trap; exit 1 ; }
trap ERROR 0
trap '{ echo "Killed by a signal"; ERROR ; }' 1 2 3 4 5 6 7 8 10 12
13 15      # list using kill -l or man kill

set -e
ecflow_client --init=$$

echo do some work
sleep 60
echo end of job

ecflow_client --complete

trap 0
exit
```

## 4 ECFLOW variables

ecFlow makes heavy use of different kinds of variables. There are several kinds of variables:

- **Environment variables** which are set in the UNIX shell before the ECFLOW-programs start. These control the server, and client (CLI).
- **Internal variables: suite definition variables.** These control server, ecflowview and CLI.
- **Generated variables:** These are generated within the suite definition node tree during job creation and are available for use in the jobs file.

This chapter lists the generated and user defined variables which have special meaning for ecFlow itself.

In an ecFlow script, ecFlow variables are written as text enclosed by a pair of %-characters (the **edit-character**.) As in C-format strings, if there are two %-characters together they are concatenated to form a single %-character in the job-file. For example if you need to execute the UNIX date command “**date +%d**”. For a job, you must enter it as “**date +%d**” into the ecFlow file.

The default **edit-character** is defined when ecFlow is compiled. It is possible to configure the **edit-character** to be defined as a variable `ECF_MICRO` (see section 3.6). The default installation uses the %-character.

You can define variables in a suite definition file using the **edit** keyword. User defined variables can occur at any node level: suite, family or task. ecFlow also **generates** variables from the node name, the host on which ecFlow is running, the time, the date and so on.

### 4.1 Variable inheritance

When a variable is needed at submission time, it is first sought in the task itself. If it is not found in the task, it is sought from the task's parent and so on, up through the node levels until found. For any node, there are two places to look for variables. The user-defined variables are looked for first, and then any generated variables.

At present, an **undefined variable** causes a task to *abort* immediately, without submitting a job. ecflowview will display this failure in the info-window.

**Example of Variable inheritance**

```

suite x
  edit TOPLEVEL 10
  edit MIDDLE 10
  edit LOWER 10
  family f
    edit MIDDLE 20
    task t
      edit LOWER abc
    task t2
  endfamily
  family f2
    edit TOPLEVEL 40
    task z
  endfamily

```

This would produce the following results:

**Table 4-1 Results of variable inheritance**

Command	task t	Task t2	task z
echo TOPLEVEL %TOPLEVEL%	10	10	40
echo MIDDLE %MIDDLE%	20	20	10
echo LOWER %LOWER%	abc	10	10

## 4.2 ECFLOW Server environment variables

ecFlow server environment variables control the execution of ecFlow and may be set before the start of server, typically in a *start-up script*.

ecFlow will start happily without any of these variables being set, since all of them have a default value. These default values can be overridden by:

- Setting them in "**server\_environment.config**". This file should then be placed in the current working directory, when invoking the server.
- Explicitly setting environment variables. These will override any setting in the "**server\_environment.config**" file.

Table 4-2 ECFLOW environment variables

Variable name	Explanation	Default value if variable not set
ECF_HOME	Home for all the ecFlow files Different meaning for ecFlow itself and suites	Current working directory
ECF_PORT	Server port number	3141 (default but customisable)
ECF_JOB_CMD	Command to be executed to send a job	%ECF_JOB% 1> %ECF_JOBOUT% 2>&1
ECF_CHECK	Name of the checkpoint file	ecf.check
ECF_CHECKOLD	Name of the backup of the checkpoint file	ecf.check.b
ECF_LOG	History, or log file	ecf.log
ECF_CHECKINTERVAL	The interval to save the check point file	120
ECF_LISTS	ecFlow white-list file. Controls read write access to the server for each user	ecf.lists
ECF_SERVERS	ecFlow and CLI nickname table	/usr/local/lib/ecflowview/servers

### 4.3 Environment variables for the ECFLOW client

Some of these variables must be set and exported before any of the **ecflow\_client** commands are executed. Since the script/job can call **ecflow\_client**, then typically they are all set in an include file in the header of the task so that all tasks would have them correctly set. Table 4-3 shows environment variables used by the ecFlow client.

Table 4-3 Environment variables for the ECFLOW client

Variable name	Explanation	Compulsory?	Example
ECF_NODE	Name of the host running ECF	Yes	hostname[.domain.name] nickname

Variable name	Explanation	Compulsory?	Example
ECF_NAME	Full name of the task	Yes	/suite/family/task
ECF_PASS	Jobs password	Yes	(generated)
ECF_PORT	port number	No	3141, This must match server port number
ECF_TRYNO	Task try number	No	(generated)
ECF_HOSTFILE	File to list possible alternate ECFs	No	/home/user/avi/.ecfhostfile
ECF_TIMEOUT	Maximum time in second for the client to try to deliver the message	No	1 - 86400
ECF_DENIED	If set to 1 and ecFlow denies access, the client will exit with failure	No	1

#### 4.4 ECFLOW suite definition variables

The suite definition variables are created like:

```
edit VAR 'the name of the variable'
```

They can also be created via the python API.

Any user created variable take precedence over suite definition variable of the same name.

These suite definition variables control the execution of ECF. Defining these variables you can, for example, control how a job is run, how ecFlow files are located or where the job output should go. Table 4-4 shows a list of ecFlow variables.

Table 4-4 ecFlow variables

Variable name	Explanation	Default value exists	Example
ECF_KILL_CMD	Method to kill a running task. Depends on how task was submitted via <a href="#">ECF_JOB_CMD</a> . ecFlow must know the value of remote id ( <a href="#">ECF_RID</a> )	No	rsh %SCHOST% qdel -2 %ECF_RID% > %ECF_JOB% 2>&1

Variable name	Explanation	Default value exists	Example
	.Variable enables <code>kill(CLI)</code> command to be used. Can use the generated variable <code>%ECF_JOB%.kill</code> for storing command output		
ECF_JOB_CMD	Command to be executed to submit a job. May involve using a <i>queuing system</i> , like NQS, or may running the job in the background.	Yes	<code>%ECF_JOB% 1&gt; %ECF_JOBOUT% 2&gt;&amp;1</code> <code>%SCHOSt%submit %ECF_JOB%</code>
ECF_STATUS_CMD	Command to be used to check the status of a submitted or running job. Can use the generated variable <code>%ECF_JOB%.stat</code> for storing command output	No	<code>'rsh %SCHOSt% qstat -f %ECF_RID%.%SCHOSt% %ECF_JOB% 2&gt;&amp;1'</code>
ECF_URLCMD	Command to be executed to allow user to view related web pages	No	<code>`\${BROWSER:=firefox} -remote 'openURL(%ECF_URLBASE%/%ECF_URL%)'</code> Where <code>ECF_URLBASE</code> is the base web address and <code>ECF_URL</code> the specific page.
ECF_HOME	The default location for ecFlow files if <code>ECF_FILES</code> is <b>not</b> used. The location for <i>generated files</i> . These are the job-files and the job-output. Setting this variable to a different directory to <code>ECF_FILES</code> enables you to <i>clean up</i> all the files produced by running ECF.	Yes	<code>/tmp/ECF/`\${SUITE}</code>
ECF_TRIES	Number of times a job should rerun if it aborts. If more than one and the job aborts, the job is automatically <i>re-run</i> by ECF. Useful when jobs are run in unreliable environments. For example using commands like <code>ftp(1)</code> in a job can fail easily, but re-running the job will often work.	Yes	2

Variable name	Explanation	Default value exists	Example
ECF_FILES	Alternate location for ecFlow files	No	/home/user/ECF/\$SUITE
ECF_INCLUDE	Path for the include files.	No	/home/user/ECF/\$SUITE/include
ECF_EXTN	Overrides the default script extension	Yes	.sms (default is .ecf)
ECF_DUMMY_TASK	Some tasks have no associated '.ecf' file. The addition of this variable stops job generation checking from raising errors.	No	Any value is sufficient
ECF_OUT	Alternate location for job and cmd output files. If this variable exists it is used as a base for <code>ECF_JOBOUT</code> but it is also used to search for the output by ecFlow when asked by ecflowview/CLI. If the output is in <code>ECF_HOME/ECF_NODE.ECF_TRYNO</code> it is returned, otherwise <code>ECF_OUT/ECF_NODE.ECF_TRYNO</code> that is <code>ECF_JOBOUT</code> is used. Job may continue to use <code>ECF_JOBOUT</code> (as in a QSUB directive) but should copy its own output file back into <code>ECF_HOME/ECF_NODE.ECF_TRYNO</code> in the end of their run.	No	/scratch/ECF/
ECF_MICRO	ecFlow <i>pre-process character</i> to be used by ecFlow pre-processor for variable substitution and including files.	Yes	%

## 4.5 Generated Variables

ecFlow generates time and date variables in various formats from the clock. There is a separate clock for every suite. Scripts can make use of these generated variables. The variables are available at the suite level and may be *overridden* by an `edit` keyword at the suite, family or task level. In ecflowview generated variables are bracketed, e.g. (ECF\_TRYNO = 0).

These variables are *generated* by ecFlow from the information in the definition file and are available for use in ecFlow files. Normally there is no need to *override* the value by using **edit** statement in the definition file. Table 4-5 shows a list of generated variables.

Table 4-5 Generated variables

Defined for	Variable name	Explanation	Example
super or root	ECF_TRIES	The default number of tries for each task	2
	ECF_PORT	The port number	3141
	ECF_NODE	The hostname of the machine running ECF	host_1
	ECF_HOME	Home for all the ecFlow files	\$CWD
	ECF_JOB_CMD	Command to be executed to send a job	%ECF_JOB% 1> %ECF_JOBOUT% 2>&1
	ECF_LISTS	Name of the ecFlow white-list file	ecf.lists
	ECF_PASS	Default password string to replace the real password, when communicating with the server.	_DJP_
	ECF_LOG	Name of the ecFlow history, or log file	ecf.log
	ECF_CHECK	Name of the checkpoint file	ecf.check
	ECF_CHECKOLD	Name of the backup of the checkpoint file	ecf.check.b
Suite	SUITE	The name of the suite	Backarc
	DATE	Date of the suite in format DD.MM.YYYY	21.02.2012
	DAY	Full name of weekday	Sunday
	DD	Day of the month, with two digits	07

Defined for	Variable name	Explanation	Example
	DOW	Day Of the Week	0
	DOY	Day Of the Year	52
	MM	Month of the year, with two digits	02
	MONTH	Full name of the month	February
	YYYY	Year with four digits	2012
	ECF_DATE	Single date in format YYYYMMDD	20120221
	ECF_TIME	Time of the suites clock, HH:MM	20:32
	ECF_CLOCK	Composite weekday, month, Day Of the Week, Day Of Year	sunday:february:0:52
Family	FAMILY	The name of the family, (avoid using)	get/ocean
	FAMILY1	The last part of the family, (avoid using)	Ocean
Task	TASK	The name of the task	Getobs
	ECF_RID	The Request ID of the job (only for running jobs)	PID
	ECF_TRYNO	The current try number for the task. After <b>begin</b> it is 1. Number is advanced if the job is re-run. Used for in output and job-file numbering. Since this variable must be numeric and is used in the file name generation, it should <b>not</b> be defined by the user.	1
	ECF_NAME	Full name of the task	/backarc/get/getobs/getobs
	ECF_PASS	Password for the task to enable login to ECF	xyZ12Abx
	ECF_SCRIPT	The full pathname for the script (if <b>ECFFILES</b> was not used) The	/home/ma/map/ECF/back/

Defined for	Variable name	Explanation	Example
		variable is composed as <code>ECF_HOME/ECF_NAME.ecf,</code>	<code>get/getobs/ getobs.ecf</code>
	ECF_JOB	Name of the job file created by ECF. The variable is composed as <code>ECF_HOME/ECF_NAME.jobECF_T RYNO.</code>	<code>/some/path/ back/get/ getobs/ getobs.job1</code>
	ECF_JOBOUT	Filename of the jobs output. ecFlow makes the directory from <code>ECF_HOME</code> down to the last level. The variable is composed as <code>ECF_HOME/ECF_NAME.ECF_TRYN O.</code>	<code>/some/path/ back/get/ getobs/ getobs.1</code>

**Note for a suite:** There are many variables derived from the `clock` of the suite. It is planned to remove all suite variables mentioned apart from those with the ECF prefix and `SUITE`.

**Note for a family:** For the variable `FAMILY` the value is generated from each family name by adding a slash, '/', in between.

**Note for a task:** The password exists only at submission time. During jobs execution, only the encrypted password is available in ECF. If a task does not have a variable `ECF_PASS`, ecFlow generates one. This is the only variable which is **not** searched in the normal way.

## 4.6 Variables and Substitution

This section describes how to use suite definition **variables** in ecFlow files. Suite definition variables are defined by the “edit” keyword. The section on “ecFlow pre-processor” gives more information on how variables are used.

In an ecFlow script, variables are written as text enclosed by a pair of '%' characters (the *edit-character* or *micro-character*). As in C-format strings, if there are two %-characters together they are concatenated to form a single %-character in the job-file. For example if you need to execute UNIX command

```
date +%d
```

In a job, you must enter it as following into an ecFlow file:

```
date +%d
```

At present, the default edit-character is %. It can only be defined when ecFlow is compiled. It can be redefined by setting the variable `ECF_MICRO`.

A user defines variables in a suite definition file using the `edit` keyword. User defined variables can occur at any node level: suite, family or task. ecFlow also *generates* variables from the node name, the host on which ecFlow is running, the time, the date and so on.

When a variable is needed at submission time, it is first sought in the task itself. If it is not found in the task, it is sought from the task's parent and so on, up through the node levels until found. For any node, variables are looked for in the following order:

- The user-defined variables are looked for first
- Repeat name, in which case current repeat value is used
- Finally generated variables.

An undefined variable causes a task to abort immediately, without the job being submitted. A *flag* is set in the task and an entry is written into ECF-logfile. If this is too severe you can use default variables in your scripts

```
%VAR:value%
```

If variable "VAR" is not found, then we use a default value of "value"

Clever use of variables can, however, save a lot of work. For example you can use the same script in multiple places, but configure it to behave differently depending on the variable set.

#### 4.6.1 ECFTRIES and ECFTRYNO

If you have set ECFTRIES in your definition file to be greater than one then your task will automatically rerun on an abort. You can then use the ecFlow variable ECF\_TRYNO to modify the behaviour of your tasks dependant on the try number, e.g.

```
# QSUB -o %ECFBASE%/log%ECFNAME%.%ECF_TRYNO%

if [ %ECF_TRYNO% -gt 1 ] ; then
    DEBUG=yes
else
    DEBUG=no
fi
```

## 5 Text based suite definition format

ecFlow manages suites. A suite is a collection of families, and a family is a collection of tasks and possibly other families. Tasks may have events, meters, labels etc. When it does not matter which one of the terms suite, family or task is in question, the generic term *node* refers to any of them.

By default, suites are independent of each other. If necessary, suites can have *cross-suite dependencies*. These are best avoided, since ideally a suite is a self-contained unit.

There is an analogy between a suite definition and the UNIX file system hierarchy (see Table 5-1).

Table 5-1 Suite definition analogy with UNIX file system

<b>Suite</b>	File system
<b>Family</b>	Directory
<b>Task</b>	File (executable)
<b>Event</b>	Signal (not part of the file system)
<b>Meter</b>	Numerical signal (not part of the file system)
<b>Label</b>	Text signal (not part of the file system)
<b>Dependency</b>	Soft link (can span file systems)

Normally a suite is defined using a file, or via the python API. Suite definitions are best stored a suite per file.

A suite definition is normally placed in a definition file. Typically the name for the file is *suitename.def*, but any name may be used.

It is good practice to list all the attributes for families and suites before any tasks are defined. This makes the reading of the suite-definition file easier.

There are two ways of defining a suite.

- Using the ASCII suite definition file format
- Using the ecFlow python API

## 5.1 Defining a suite, using the text definition file format

This section describes how to create nodes in a suite. There are some limitations on the use of this functionality. **I.e. conditionals and looping structures are not allowed.**

Families can only exist inside a suite or inside other families.

### 5.1.1 suite

The **suite** keyword is used to start a new suite definition. There can be only one suite defined in a definition file.

The only parameter the **suite** command takes is the name of the new suite to be defined. After this command, all other commands define something in the suite. Currently, there cannot be dependencies at the suite level.

A suite is a collection of families, variables, repeat and clock definitions.

A suite is the only component that can be started using **begin (CLI)**

```
suite x
  clock hybrid
  edit ECF_HOME "/some/other/dir"
  family f
  ...
endsuite
```

### 5.1.2 family

The **family** keyword is used to create a new family inside a suite or inside another family. The only implicit action is to terminate the previous task definition.

The only parameter this keyword takes is the name of the new family to be defined.

The definition of a family must be terminated by either **endfamily** or **endsuite** or by the end of the suite definition file.

A family is used to collect tasks together or to group other families. Typically you place tasks that are related to each other inside the same family, analogous to the way you create directories to contain related files.

Sometimes it is useful to group a set of tasks into a family to get the trigger dependencies cleaner, e.g. you might have ten tasks that all need to be complete before the eleventh task can run, as in the following definition file.

```
family f
  task t0
  family ff
    task t1
    ...
    task t10
  endfamily
  task t11
  trigger ff==complete
endfamily
```

### 5.1.3 task

This keyword defines a new task inside a family or suite. The only parameter that this keyword takes is the name of the new task to be defined. This keyword acts as an implicit **endtask** for the previous task.

Only tasks can be submitted. A job inside a task script should generally be re-entrant so that no harm is done by rerunning it, since a task may be automatically submitted more than once if it aborts.

The ability to restart is important. The idea of using ecFlow is to divide a real problem into small manageable parts each handled by a separate task or family. On occasion, especially when events are used, a task should be able to start from the point at which a previous job finished. Events should be sent only once, although there is no harm in sending them more than once. A typical example of a large real problem is a weather forecast which may take several hours to run, check-pointing itself from time to time. If the forecast fails and is restarted, it can determine how far it had already progressed and continue from where it left off.

### 5.1.4 event

The event keyword assigns an event to the task currently being defined. Only tasks can have events and they can be considered as an attribute of a task. There can be many events and they are displayed as nodes.

An event has a number and possibly a name. If it is only defined as a number, its name is the text representation of the number without leading zeroes. For example, event 007 can be accessed as event 7 or as event 007. Event's numbers must be positive and their name can contain only letters and digits. The use of letters is optional; the event name can consist simply of digits.

If a name is given, you can only refer to the event by its name, not by number. As such there is no point in using a number and a name, unless they are the same:

```
task x
  event 1                # Can only be referred to as x:1
  event 2 prepok         # Can only be referred to as x:prepok
  event 3 99             # This is asking for trouble!
```

When a job sends an event, it is saying that part of its task has been carried out and that any task waiting for that part can now start, unless it also needs other conditions to be met. If the job then aborts and the task is resubmitted, the restarted job should be able to carry on from where it previously left off. Otherwise, there is the possibility of destroying information needed by the task triggered by the event.

In order to use events you have to first define the event in the suite definition file, e.g.

```
suite x
  family f
    task t
      event foo
```

Where 'foo' is the name of the event. The default value for event is "clear" or false (the value shown when the suite begins). After (command begin) it looks like:



Then you can modify your task to change this event while the job is running, e.g.

```
ecflow_client --init $$
ecflow_client --event foo
ecflow_client --complete
```

After the job has modified the event it looks like:



Now the value of the event is "set" or true.

### *Using events in triggers*

The purpose of an event is to signal partial completion of a task and to be able to trigger another job which is waiting this partial completion. Task "t1" creates a file, sets an event and saves the file (which might take a long time.) Another task, "t2" only needs the on-line copy of the file so it can start as soon as the file is made, e.g.

```

suite x
  family f
    task t1
      event foo
    task t2
      trigger t1:foo == set

```

The " = set" part is optional since the value of the event is Boolean anyway.

### 5.1.5 Meter

This is an extension to the event. In some tasks there may be many events which are set in order, e.g. in a 10 day weather forecast an event might be set every six hours, more than 40 events. These events are set in increasing order. By creating a meter that can have values, in this case from 0 to 240, will help to have more valid information on the display.

In a suite definition file one would have

```

task forecast
  meter step 0 240 240
  # meter name min max [colour-change]

```

The meter can be used in triggers in the same way as the events, except that the meter will have a value, e.g.

```

task plot5days
  trigger fc/model:step eq 120 # 5 days done

```

The numeric value used in the triggering means that there is an ambiguity if you have a node with the same name, let's say task "120". Earlier versions of SMS required that names start with a letter. (This rule was relaxed with a warning in 4.1 and 4.2 didn't even issue a warning message).

#### Using meters

In order to use meters you have to first define the meter in the suite definition file, e.g.

```

suite x
  family f
    task t
      meter foo 0 100 100

```

foo is the "name" of the meter and the three numbers are minimum, maximum, and threshold values for the meter. The default value is the minimum value (the value show when the suite begins). After the command "begin" it looks like:



In the ecFlow job file you can then modify your task to change the meter while the job is running, e.g. like:

```
ecflow_client --init $$  
for i in 10 20 30 40 ... ; do  
    ecflow_client --meter foo $i  
    sleep 1  
done  
ecflow_client --complete
```

After the job has modified the meter a few times it looks like:



And in the end the meter looks like:



### *Using meters in triggers*

The purpose of a meter is to signal proportional completion of a task and to be able to trigger another job which is waiting this proportional completion. Let us say that task "model" creates a hundred files, and there are ten other tasks to process these files. Task "t0" processes files 0-9, task "t1" files 10-19 and so on. The python API would look something like:

```
suite = Suite("x")  
f = suite.add_family("f")  
task_model = f.add_task("model")  
task_model.add_meter("file",0,100,100)  
for i in range(0,9):  
    file = i*10 + 10  
    t = f.add_task("t" + str(i))  
    t.add_trigger("model:file ge " + str(file))
```

### **5.1.6 label**

A label has a name and a value and is a way of displaying information in ecflowview. Since the value can be anything (ASCII) it cannot be used in triggers.

```

task x
  label name string
  label OBS 0
  label file "" # for empty label

```

The value of the label is set to be the default value given in the definition file when the suite is begun. This is useful in repeated suites: a task sets the label to be something, e.g. the number of observations, and once the suite is complete (and the next day starts) the number of observations is cleared.

### Using labels

In order to use labels you have to first define the label in the suite definition file, e.g.

```

suite x
  family f
    task t
      label foo ""

```

foo is the "name" of the label and the empty string is the default value of the label (the value shown when the suite begins). After the command begins it looks like:



In an ecFlow job file you can then modify your task to change the label while the job is running, e.g.

```

ecflow_client --init $$
ecflow_client --label=foo "some text"
ecflow_client --complete

```

After the job has modified the label it looks like:



If you want to send more than one line, use spaces in the text, e.g.

```

ecflow_client --init $$
ecflow_client --label=foo multi line label
ecflow_client --complete

```



And to have the display lined up better, send the first line as empty:

```
ecflow_client --init $$  
ecflow_client --label "" multi line label  
ecflow_client --complete
```



### 5.1.7 limit

This command defines a limit into the current node. It is a means of providing simple load management by say limiting the number of tasks submitted to a specific server.

Typically you either define limits on suite level or define a separate suite to hold limits so that they can be used by multiple suites.

```
suite limits  
  limit sgi 10  
  limit mars 10  
endsuite  
  
suite obs  
  family limits  
    limit hpcd 20  
  endfamily  
  extern /limits:sgi  
  task t1  
    inlimit sgi  
  task t2  
    inlimit /obs/limits:hpcd  
endsuite
```

### *Using limits*

In order to use limits you have to first define a limit in the suite definition file and then you must also assign families/tasks to use this limit, e.g.

```
suite x  
  limit fast 1  
  family f
```

```
inlimit /x:fast
task t1
task t2
```

Where fast is the "name" of the limit and the number after the name defines maximum number of tasks that can run simultaneously using this limit. That's why you do not need a trigger between tasks "t1" and "t2".

There is no need to change the tasks. The jobs are run in the order they appear in the suite definition. Be aware that command order may be used to modify the order after the suite definition has been defined.

The following is a sequence if the jobs run in the normal order



And if you manually now re-run the "t1" (you go over the limit!)



Using integer limits is a bit different. In the following example we define limit disk to be 50 (megabytes) and task using 20 (megabytes) each. This means that only two of them can be running any given time.

```
suite x
  limit disk 50
  family f
    inlimit disk 20
    task t1
    task t2
    task t3
```

Where disk is the "name" of the limit, 50 is the maximum value this limit can be.

### 5.1.8 endsuite

This terminates a suite definition. **endsuite** is also an implicit **endfamily/endtask** for all families/tasks currently being defined.

### 5.1.9 endfamily

This terminates a family definition. You must use **endfamily** to terminate the current family in order to start a new family definition at the same level.

Typically **endfamily** is followed by a **task**, a **family** or an **endsuite** command.

#### 5.1.10 endtask

This terminates a task definition. This is not strictly speaking needed, unless you want to add properties to the family containing the task or if you are using automatic generation to setup a suite. The example below highlights the use of **endtask**

```
family f
  task t1
  task t2
    edit ECFHOST c90
  endtask
  edit ECFHOST ymp8    # This variable is for family f!
endfamily
```

The following achieves the same effect and is clearer:

```
family f
  edit ECFHOST ymp8
  task t1
  task t2
    edit ECFHOST c90
endfamily
```

## 5.2 Dependencies

The following section describes commands that are used to create dependencies between nodes in a suite.

A node can be made dependent in following ways: time/date, other node(s) or limited from running by some resource. There can be multiple time dependencies which may be expressed using the time of day, the day of the week, or the date. A node dependency is

expressed as a logical statement about another node and its state, like `taskname == complete`. A dependency may involve several other nodes, preferably all in the same suite.

A node that is dependent cannot be started as long as some dependency is *holding* it. For triggers, the phrase *trigger is set* means a trigger has *expired*, and *trigger is not set* means it is still *holding*. By default, a node depends on its parent. So, for example, a task cannot start if the family to which it belongs is still waiting on a dependency.

A node can have many time dependencies, but only one, albeit complex, trigger. When a suite begins, the trigger and all the time dependencies hold the node. The node stays queued as long as the trigger is not set. Only one of the dependency types (time, date or day, and trigger) can *expire* at a time, the others still remain holding or in use. If the trigger is set, one of the possible time dependencies may expire and let the node go. When the node completes, the expired time dependency is marked as being used, and the other two time dependencies are processed.

The node can be dependent because:

- Server is halted or shutdown
- Its parent is dependent
- It is triggered by a state of another node
- It is waiting for time of day
- It is waiting for date of year
- It is waiting for day of a week
- Limit it uses does not have a free token
- It is migrated (restored at begin)
- It is suspended

The following sections discuss the different dependency types, and give examples of how to use them together.

### 5.2.1 trigger

This defines a dependency for a task or family. There can be only one trigger dependency per node, but that can be a complex boolean expression of the status of several nodes. Triggers should be avoided on suites.

A node with a trigger can only be activated when its trigger has expired. A trigger holds the node as long as the trigger's expression evaluation returns false. There are a few additional keywords and some names may point to other nodes with their value acting as the status of

those nodes. Trigger mathematics are computed, using double arithmetic (with no string comparisons). There should not be any need to use numerical expressions, instead logical functions (and, or, not, eq, ne) with node names should be used.

The keywords in trigger mathematics are: *unknown*, *suspended*, *complete*, *queued*, *submitted*, *active* and *aborted* for task and family status; and *clear* and *set* for event status. These keywords are treated as numbers starting from 0 (unknown) to 6 (aborted). There is no need to be aware of the numerical values as long as you do not use a trigger in the form:

```
trigger plain_name                # WARNING! DO NOT USE!
```

This is true only as long as the status of plain\_name is unknown. It is not advisable to use mathematical function names for node names.

The *full name* or *relative name* of a node can also be used as an operand. A full name starts from the super-node. A *relative name* can include "../" to indicate the parent node level. A relative name can also include "./" to indicate the same level, this is needed if the task name is numeric (otherwise its numeric value would be used in the expression.)

```
family foo
  task bar
  task foobar
endfamily
family second
  task 00z
  trigger ../foo/foobar==complete # task from previous family
  task another
  trigger ./00z == complete       # the previous task
```

For events it is convenient to use a plain name, since an event can only have values *clear* or *set*, numerically 0 or 1. So triggers of the form:

```
trigger taskname:event
trigger taskname:event == set
```

Will hold as long as the event is not set. The second line shows a clearer alternative equivalent way to write the same trigger.

Meters can be used in triggers the same as events, except that their value should be compared against numerical expression. It is important to remember to use *greater or equal* instead of *equals*. In the following example **foobar** will not be submitted if, let's say, suite is suspended while **foo** sets it meter to first 120 and then to 130. **bar** will still be submitted once the suite is resumed.

```
task foo
  meter hour 0 240
task bar
  trigger foo:hour >= 120
task foobar
  trigger foo:hour == 120          # dangerous !!!
```

There is no automatic checking for *deadlocks*, which can be difficult to detect. However, if your suite is known to complete (i.e. it does not run forever), then simulation can be used to check for deadlocks. The following example is a simple case:

```
task a ; trigger ./b == complete
task b ; trigger ./a == complete # DEADLOCKS tasks a & b
```

There is no automatic simplification of the mathematics. ecFlow will read the whole of a suite definition into memory, but with comment lines removed and possibly different indentation.

Mathematical expressions must be given in a single logical line. Use continuation lines for long expressions. For example:

```
trigger /suite/family1/task1==complete and ( /suite/family2 \
      eq complete or /suite/family3 eq complete )
```

There cannot be any characters after the *line continuation character* `\  
any keyword can appear in an expression but they must be used in a way that makes sense. For example, a task can *never* be *set* or *clear* and, likewise, an event can only be *set* or *clear*.

See section 5.3.6 for details on using triggers external to the suite.

### 5.2.2 date

This defines a date dependency for a node. There can be multiple date dependencies. The European format is used for dates, which is: **dd.mm.yy** as in 31.12.2012. Any of the three number fields can be expressed with a wildcard `\*` to mean any valid value. Thus, 01.\*.\* means the first day of every month of every year.

Currently, you cannot specify a range of values for any of the three number fields in a date, See “day” for a way to specify the first seven days.

```
task x          # Run the task twice a
  date 1.*.*    # month, on 1st and 15th
  date 15.*.*
```

Because ecFlow was designed with ECMWF suites in mind, the date is a very important notion. ecFlow defines the time using clocks. A clock is an attribute of a suite. Different suites can have different clocks. There are two kinds of clocks:

- Real clocks: A suite using a real clock will have its clock matching the clock of the machine.
- Hybrid clocks: A hybrid clock is a complex notion: the date and time are not connected. The date has a fixed value during the complete execution of the suite. This will be mainly used in cases where the suite does not complete in less than 24 hours. This guarantees that all the tasks of this suite are using the same date. On the other hand, the time follows the time of the machine.

Once a suite is complete, it is repeated automatically, with the next date. The value of the date is contained in the ecFlow variable ECF\_DATE, and the value of the time is in ECF\_TIME. ECF\_CLOCK contains other information such as the day of week. A job should always use the ecFlow variables, and not directly access the system date.

If a hybrid clock is defined for a suite, any node held by a date dependency will be set to complete at the beginning of the suite, without the node ever being dispatched. Otherwise the suite would never complete.

### 5.2.3 day

This defines a day dependency for a current node. The parameters are the names of weekdays (lowercase): *sunday, monday, tuesday, wednesday, thursday, friday* or *saturday*. Any combination is acceptable. Names must be typed in full and at least one must be given. Giving the same weekday more than once is not treated as an error. The names can be shortened as long as there is no ambiguity.

There can be multiple day dependencies, but it is more convenient to define just one dependency, listing the weekdays the node is to run.

Combined with a date, you can specify more particular dependencies such as the first Monday in a month:

```
task x
  date 1-7.*.*          # This will not work
  day monday
  #
  # Will be listed by the show as
task x
  date 1.*.*
  date 2.*.*
```

```
...
date 7.*.*
day monday
```

Since the number of days in a month varies, there is no direct means to specify, say, every last Sunday of each month. A list of dates must be provided.

If a hybrid clock is defined, any node held by a day dependency will be set to complete at the beginning of the suite, without the node ever being dispatched. Otherwise the suite would never complete.

#### 5.2.4 time

This defines a time dependency for a node. Time is expressed in the format `[h]h:mm`. Only numeric values are allowed. There can be multiple time dependencies for a node, but overlapping times may cause unexpected results.

To define a series of times, specify the start time, end time and a time increment.

If the start time begins with '+', times are relative to the beginning of the suite or, in repeated families, relative to the beginning of the repeated family.

If relative times are being used, the end time is also relative.

```
time 15:00          # at 15:00
time 10:00 20:00 01:00 # every hour from 10am to 8pm
time +00:01         # one minute after the suite begins
time +00:10 01:00 00:05 # 10-60 min after begin, every 5 min
```

There is no direct way to specify that a node should be submitted on different days at different times.

To get a task to run at two specific times you can use two separate time commands.

```
task t1
  time 15:00          # run at 15:00
  time 19:00          # also run at 19:00
```

**Note:** you should take care with tasks using the time command to cause the suite to cycle on fast systems. If the task takes less than a minute to run then there is a possibility that the trigger will still be valid once the suite has cycled. This can be avoided by making sure that such tasks take longer than one minute to run, for example, by adding a sleep command.

### 5.2.5 today

Like "time", but "**today**" does not wrap to tomorrow. If suites' begin time is past the time given for the "today" command the node is free to run (as far as the time dependency is concern.)

```
today 3:00                # today at 3:00
today 10:00 20:00 01:00   # every hour from 10am to 8pm
```

This is useful when you have early morning time-dependency, but you want repeat the suite in the afternoon. Otherwise the node would wait till the next morning to run.

### 5.2.6 cron

Like **time**, **cron** defines time dependency for a node, but it ignores the suites **clock** and allows the node to be **repeated** indefinitely.

This means that nodes, and thus suites with **cron** will never complete.

```
cron 23:00                # run every day at 23:00
cron 10:00 20:00 01:00   # run every hour between 10am and 8pm
cron -w 0,1 10:00        # run every Sunday and Monday at 10am
cron -d 10,11,12 12:00   # run 10th, 11th and 12th of each month ~
                        # at noon
cron -m 1,2,3 12:00      # run on January, February and March
                        # every day at noon
```

```
cron -w 0 -m 5,6,7,8 10:00 20:00 01:00
```

```
task x
```

```
  cron -w 1,2,3,4,5 10:00
  cron -w 0,6 12:00
```

```
task x
```

```
  cron -w
```

When the node becomes complete it will be queued immediately. This means that the suite will never complete, and output is not directly accessible through ecflowview

If tasks abort, ecFlow will not schedule it again. Also if the time the job takes to complete is longer than the interval a "slot" is missed, e.g. **cron 10:00 20:00 01:00** if the **10:00** run takes more than an hour the **11:00** run will never occur.

With cron you can also specify weekdays, day of the month and month of the year masks.

### 5.2.7 Using dependencies together

The way a combination of different dependencies work together is not always clear. In order for a node to be scheduled:

- **Parent:** must be free in order for its children to run. A family must be free before any of its tasks can run.
- **Date:** must be free. This is checked when you begin a suite `begin(CLI)` and at midnight for the suite. There is no midnight for a hybrid clock.
- **Time:** must be free. This is checked every minute (configurable.)
- **Trigger:** must be free. This is checked every time there is a state change in the suite for all potential nodes.
- **Limit:** must not be full. Boolean limits just have slots in them, while integer limits must have enough space for the usage.

Here are a few examples of combinations with their behaviour.

To run task **x** only once, on 17th of February 2012 at 10 am:

```
task x
  time 10:00
  date 17.2.2012
```

To run task **x** twice, at 10am and 8pm, on both 17th and 19th of February 2012, that is, four times in all. Notice the task is queued in between, and completes only after the last run. Under a hybrid clock, the task is run (twice) only if the date is either 17th or 19th at the time the suite begins:

```
task x
  time 10:00 ; time 20:00
  date 17.2.2012 ; date 19.2.2012
```

To run task **x** after task **y** is complete and if the day is Monday. If the suite is using a real clock, the task waits for the following Monday (unless today is Monday) and for task **y** to be complete. Under a hybrid clock, if today is not Monday when the suite begins or is auto-restarted, task **x** is marked complete without being submitted:

```
task y
task x
  trigger ./y == complete
  day monday
```

The next example shows how to run a task after an earlier task has stopped, either by completing or aborting. It may be useful to continue after a task has tried a few times but still failed. However, this technique should only be used if there is logic somewhere to correct for the missing task. Otherwise, task **y** will fail as well.

```
task x
task y
    trigger ./x == complete or ./x == aborted
#
# The above trigger using named operators:
#
# trigger x eq complete or x eq aborted
#
task z
    trigger (x==complete or x==aborted) \
        and (y==complete or y==aborted)
```

To run a task on a series of given days, use the Python API:

```
t = ecflow.Task("x")
for i in [ 1 , 2, 4, 8, 16 ] :
    t.add_date(i,0,0) # 0 means any day,month or year

# Will be displayed by the show(CLI) command as
task x
    date 1.*.*
    date 2.*.*
    date 4.*.*
```

To run a task half after the previous task **fc** has done half of its work:

```
task fc
    meter hour 0 240
task half
    trigger fc:hour >= 120
# trigger fc:hour ge 120
```

There is no guarantee that a task will be sent at the exact moment requested. At the specified time, ecFlow might be busy processing other tasks. ecFlow does not check time dependencies constantly, but sweeps through them once a minute. This makes processing of events in ecFlow much more stable.

## 5.3 Attributes

This section describes commands that can be used to give attributes to nodes in a suite.

### 5.3.1 autocancel

Autocancel is a way to automatically delete a node which has completed. The deletion may be delayed by an amount of time in hours and minutes or expression in days. This will help maintenance of *living suites*. Notice that, if the suite is *repeated* and part of it is cancelled, that part will obviously not be run.

The node deletion is never immediate. The nodes are checked once a minute (by default) and expired autocancel-nodes are deleted.

Any node can have autocancel statement like:

```
autocancel +00:10      # Cancel 10 minutes later
autocancel 0           # Cancel immediately
autocancel 3           # Cancel three days later
```

The effect of **autocancel** is the same as if user would use:

```
ecflow_client --delete
```

This means the deleted nodes, if used to trigger other nodes, may leave a node to wait the (now missing) node. To solve this problem use a trigger like:

```
task t
  trigger node_name==complete or node_name==unknown
```

It is best not to use autocancelled nodes in the triggers.

### *Using autocancel*

Sometimes you may want to have a suite in which you incrementally add things and once these parts have served their purpose you want to dispose them.

**autocancel** is a way of automatically removing these families. Nodes with this property defined will be automatically removed by ecFlow once they become complete and the time defined has elapsed.

```
suite x
  family fam
    autocancel +05:00
    task t
  endfamily
  family ff
```

In this example family **fam** will be removed from the suite once it has been complete for more than five hours.

This is equivalent to the user issuing the CLI command

```
ecflow_client --delete=/x/fam
```

Which means that if there are other tasks dependent of **fam** or its children their triggers may never allow them to run. To guard against such situations you can use triggers that allow other nodes to disappear or that not been defined at all. This is done by using the status value unknown for undefined nodes.

```
suite x
  family fam
    autocancel +05:00
    task t
  endfamily
  family ff
    trigger fam==complete or fam==unknown
  ...
```

### 5.3.2 clock

Defines a *clock type* to be used by the suite, and specifies the clock gain factor. Only suites can have a defined clock. A clock always runs in phase with the system clock (UTC in UNIX) but can have any offset from the system clock. ecFlow generates variables from the current time.

The clock must be either *hybrid* or *real*. Under a hybrid clock, the date never changes unless specifically altered or unless the suite restarts, either automatically or from a **begin** command. Under a real clock, the date advances by one day at midnight. Time and date dependencies work a little differently under the two clocks. The default clock type is *hybrid*.

Clock gain is expressed in seconds and can be given as an integer, a time or a date. Seconds and time can have a sign:

```
clock real 300           # the clock gains 300 sec from now
clock real +01:00       # the clock gains 3600 sec from now
clock real 01:00        # clock is 01:00 in the morning
clock real 20.1.2012    # many days late but H:M is ok
clock real 20.1.2012 +01:00 # many days late, time gains a hour
```

The clock can only be modified using alter command, e.g.

```
ecflow_client --alter=change clock_type real /suite
```

### 5.3.3 complete

Force a node to be complete if the trigger evaluates, without running any of the nodes.

This allows you to have tasks in the suite which a run only if others fail. In practice the node would need to have a trigger also.

This allows you to have *standby* nodes which may run depending on the success of other nodes. Here is an example where task **tt** does not run if task **t** meter is more than 120. If task **t** however completes, but the meter **step** is less that **120** the *standby* job will run.

```
family f
  task t
    meter step 0 240 120
  task tt
    complete t:step ge 120
    trigger t==complete
endfamily
```

#### Using complete

The **complete** attribute can be used to *force* things to a complete state if they are queued and the condition is met.

Here is an example when we have a task that decides that the whole family should not run. As it has **repeat** in it, it will advance the repeat to the next date and run that:

Conditional complete

```
suite x
  family f
    repeat date YMD 20120601 20200531
    complete ./f/check:nofiles
    task check
      event 1 nofiles
    task t1 ; trigger check==complete
    task t2 ; trigger t2==complete
  endfamily
```

Here is a python example where we create a simple reusable experimental meteorological suite. There is a *configuration section* for the dates and synoptic cycles to be selected, and there is a *function* (add\_complete()) to select the contents for the complete statement. This

is needed since you can only have one **complete** statement for any node. The main loop of the suite is pretty straightforward.

```
class ExperimentalSuite(object):
    def __init__(self,start,end) :
        self.start_ = start
        self.end_ = end
        self.start_cycle_ = 12
        self.end_cycle_ = 12
    def generate(self) :
        suite = Suite("x")
        make_fam = suite.add_family("make")
        make_fam.add_task("build")
        make_fam.add_task("more_work")
        main_fam = suite.add_family("main")
        main_fam.add_repeat( RepeatDate("YMD",self.start_,self.end_) )
        main_fam.add_trigger( "make == complete" )
        previous = 0
        for FAM in ( 0, 6, 12, 18 ) :
            fam_fam = suite.add_family(str(FAM))
            if FAM > 0 :
                fam_fam.add_trigger( "." + str(previous) + " == complete " )
                self.add_complete(fam_fam,FAM)
                fam_fam.add_task("run")
                fam_fam.add_task("run_more").add_trigger( "run == complete")
                previous = FAM
        return suite
    def add_complete(self,family,fam):
        if fam < self.start_cycle_ and fam > self.end_cycle_ :
            family.add_complete( "../main:YMD eq " + str(self.start_) + " or ../main:YMD ge " +
str(self.end_) )
        elif fam < self.start_cycle_ :
            family.add_complete( "../main:YMD eq " + str(self.start_) )
        elif fam > self.end_cycle_ :
            family.add_complete( "../main:YMD ge " + str(self.end_) )
        return
print str( ExperimentalSuite(20120601,20120605).generate() )
```

### 5.3.4 defstatus

Defines the default status for a task/family to be assigned to the node when the **begin(CLI)** command is issued. By default any node get a **queued** status when you use begin on a suite.

**defstatus** is useful in preventing suites from running automatically once begun or in setting tasks **complete** so they can be run selectively.

For suites anything other than **suspended** is void. The status of a family reflects the status of its children.

For a family defstatus can be either "**suspended**", "**queued**" or "**complete**". A family with **defstatus complete** means that all tasks and families are marked complete without running them.

```
family f
  task t1
  task t2
    defstatus complete      # by default will not be run
  task t3
    defstatus suspended    # needs 2 B resumed
```

### 5.3.5 edit

This defines a variable for ecFlow job substitution in a node, equivalent to an external variable. There can be any number of variables. The variables are names inside a pair of ``%(ECF_MICRO)` characters in an ecFlow script. However, remember that ecFlow is case sensitive.

The content of a variable replaces the variable name in the ecFlow script at submission time. Special characters in a definition must be placed inside single quotes if misinterpretation is to be avoided or inside double quotes variable substitution is to be carried out. Quotes are needed if defining as a list.

#### Examples

```
edit ECF_JOB_CMD "/bin/sh %ECF_JOB% &"
edit ECF_JOB_CMD "/usr/local/bin/qsub %ECF_JOB%"
edit ECF_JOB_CMD "rsh %ECF_HOST% sh <%ECF_JOB% 1>%ECF_JOBOUT% 2>&1"
edit KEEPLOGS no
```

Submission is done via a **system(3)** call which executes `/bin/sh`.

### 5.3.6 extern

This allows an external node to be used in a trigger expression. All nodes in triggers must be known to ecFlow by the end of the `load` command. No *cross-suite dependencies* are allowed unless the names of tasks outside the suite are declared as *external*.

An external trigger reference is considered **unknown** if it is not defined when the trigger is evaluated.

You are strongly advised to **avoid** *cross-suite dependencies*. Families and suites that depend on one another should be placed in a single suite. If you think you need *cross-suite dependencies*, you should consider merging the suites together and have each as a top-level family in the merged suite.

To run the task `/b/f/t` when suite `'a'` is not present, use the following trigger, e.g.

```
extern /a/f/t
suite b
  family f
    task t
      trigger /a/f/t == complete or /a/f/t == unknown
```

### 5.3.7 late

Define a tag for a node to be *late*. Suites cannot be late, but you can define a late tag for submitted in a suite, to be inherited by the families and tasks.

When a node is classified as being late, the only action ecFlow takes is to set a flag. ecflowview will display these alongside the node name as an icon (and optionally pop up a window). A separate list is also kept.

Table 5-2 Ways a node can be late

Status	Reason
Submitted	The time node can stay submitted (format [+]hh:mm). Submitted is always relative, so + is simply ignored, if present. If node stays submitted longer than the time specified, the late flag is set.
Active	The time of day the node must have become active (format hh:mm) If the node is still queued or submitted by the time, late flag is set.
Complete	The time node must become complete (format [+]hh:mm). If relative, time is taken from the time node became active, otherwise node must be complete by the time given.

The submitted late time is inherited from parent if not present in the node itself. That is defining `late -s` on a suite all tasks will have that value, e.g.

```
task t1
  late -s +00:15 -a 20:00 -c +02:00
```

This is interpreted as: the node can stay submitted for a maximum of 15 minutes, and it must become active by 20:00 and the runtime must not exceed 2 hours.

### 5.3.8 repeat

Any node can be *repeated* in a number of different ways. Only suites can be repeated based on the suite `clock`. The syntax is as follows

```
repeat day      step [ENDDATE]      # only for suites
repeat integer VARIABLE start end [step]
repeat enumerated VARIABLE first [second [third ...]]
repeat string  VARIABLE str1 [str2 ...]
repeat file    VARIABLE filename
repeat date    VARIABLE yyyyymmdd yyyyymmdd [delta]
```

The idea is that the variable given is advanced when the node completes and the node is re-queued (except, of course, when the variable has the last value.)

Day repeats are only available for a suite (tied to clock) in which case an ending date can be given. For this to work the clock type must be hybrid: a real-time suite cannot be stopped by means of end time.

```
repeat string INPUT str1 str2 str3
repeat integer HOUR 6 24 6
repeat date YMD 20200130 20200203
```

The following suite will run in hybrid clock for 15th until 25th (inclusive)

```
suite x
  clock hybrid 15.05.2020
  repeat day 1 25.05.2025
  ...
```

**Note:** Only four digit years are allowed. Also that `force complete` will only force the current running job to be complete, but if the repetition is not finished, the next job will be sent (with the variable advanced accordingly.)

**Tip:** we prefer to use the repeat date structure in our suites. This allows us to see more easily what date the suite is running.

## 6 Defining a suite using the python API

EcFlow provides a python API. This allows:

- complete specification of the suite definition, including trigger and time dependencies
- full access to the command level interface(CLI)

Since the full power of python is available to specify the suite definition, there is considerable flexibility. The API is documented using the python `__doc__` facility.

### 6.1 PYTHONPATH and LD\_LIBRARY\_PATH

Before the ecFlow python API can be used you need to set some variables

- PYTHONPATH must be set to include the directory where the file **ecflow.so** has been installed
- LD\_LIBRARY\_PATH must be set to include the directory where **libboost\_python.so** has been installed.

### 6.2 Defining a suite, using the Python API

This section describes how to create nodes in a suite using the Python API. This method has increased functionality over the text based format.

#### 6.2.1 Add suite, family and task

The following example shows how suites, families and tasks are added to a Python definition file.

```
import ecflow

if __name__ == "__main__":

    defs = ecFlow.Defs()           # create an empty definition

    suite = defs.add_suite("s1");   # create a suite and add it to the defs

    family = suite.add_family("f1") # create a family and add it to suite

    for i in [ "a", "b", "c" ]:     # create task ta,tb,tc

        family.add_task( "t" + i)   # create a task and add to family

    defs.save_as_defs("my.def")    # save defs to file "my.def"
```

## 6.2.2 Adding Meters, Events and Labels

```
task = ecflow.Task("t1")

task.add_event( 2 )           # event reference with 2

task.add_event("wow")       # event reference with name "wow"

task.add_event( 10,"Eventname2" ) # event referenced with name "Eventname2"

task.add_meter( "metername3",0,100 ) # name, min, max

task.add_label( "label_name4", "value" ) # name, value
```

## 6.2.3 Adding Limits and Inlimit

```
s1 = ecflow.Suite("s1");

s1.add_limit( "limitName4", 10 ) # name, maximum token

f1 = ecflow.Family("f1")

f1.add_inlimit( "limitName4", "/s1/f1", 2) # limit name, path to limit, tokens consumed
```

## 6.2.4 Adding Variables

```
s1 = ecflow.Suite("s1");

s1.add_variable("HELLO","world") # name, value

a_dict = { "name":"value", "name2":"value2", "name3":"value3", "name4":"value4" }

s1.add_variable(a_dict)
```

## 6.3 Dependencies

### 6.3.1 Adding Triggers and Complete

```
task = ecflow.Task("t1")

task.add_trigger( "t2 == active and t4 == aborted" )

task.add_complete( "t2 == complete" )
```

### 6.3.2 Adding Time Dependencies

```
task = ecflow.Task("t2") # create a task
```

```

task2.add_date( 1,2,2010 )           # day, month, year

task2.add_date( 1,0,0 )             # first of each month or every year same as: 1.*.*

task2.add_day( "monday" )

task2.add_today( 0,10 )             # hour, minute, same as today 0:10

task2.add_today( ecflow.Today( 0,59, True )) # hour, minute, relative, same as today +0:59

start = ecflow.TimeSlot(0,0)       # adding a time series

finish = ecflow.TimeSlot(23,0)

incr = ecflow.TimeSlot(0,30)

ts = ecflow.TimeSeries( start, finish, incr, True); # same as today +00:00 23:00 00:30

task2.add_today( ecflow.Today(ts) )

cron = ecflow.Cron()

cron.set_week_days( [0,1,2,3,4,5,6] )

cron.set_days_of_month( [1,2,3,4,5,6] )

cron.set_months( [1,2,3,4,5,6] )

cron.set_time_series( "+00:00 23:00 00:30" )

task2.add_cron( cron );

```

### 6.3.3 Adding Large Trigger Dependencies

Please note that after the first part trigger has been added, subsequent part triggers must include a boolean to indicate whether the part expression is to be 'anded' or 'ored'

```

task2 = ecflow.Task("t2")

task2.add_part_trigger( "t1 == complete")

task2.add_part_trigger( "t2 == active", True) # here True means add as 'AND'

task2.add_part_trigger( "t3 == active", False) # here False means add as 'OR'

# complete expression is: (t1 == complete and t2 == active or t3 == active)

```

## 6.4 Attributes

### 6.4.1 Adding Defstatus, Autocancel

```
task2 = ecflow.Task("t2")

task2.add_defstatus( ecflow.DState.complete );

task2.add_autocancel( 3 )      # 3 days

t3 = ecflow.Task("t3")

t3.add_autocancel( 20,10,True ) # hour, minutes, relative
```

### 6.4.2 Adding a Clock

```
suite = ecflow.Suite("suite");      # create a suite

clock = ecflow.Clock(1,1,2010,False) # day, month, year, hybrid

clock.set_gain(1,10,True)           # hour, minutes, bool(true ~positive gain )

suite.add_clock(clock)

s1 = ecflow.Suite("s1")              # create a different suite

clock = ecflow.Clock(1,1,2010,True)  # day, month, year, hybrid

clock.set_gain_in_seconds(300,True)

s1.add_clock(clock)
```

### 6.4.3 Adding Repeats

```
f = ecflow.Family("f")

f.add_repeat( ecflow.RepeatDate("YMD",20100111,20100115,2) )

f1 = ecflow.Family("f1")

f1.add_repeat( ecflow.RepeatInteger("count",0,100,2) )

f2 = ecflow.Family("f2")

color_list = ["red", "green", "blue" ]

f2.add_repeat(ecflow.RepeatEnumerated("enum",color_list) )

task6 = ecflow.Task("t6").add_repeat( RepeatString("R", ["a","b","c"]))
```

#### 6.4.4 Adding a Late attribute

```
late = ecflow.Late()

late.submitted( 20,10 )    # hour, min

late.active( 2, 10 )      # hour, min

late.complete( 3, 10, True) # hour, min, relative

t1 = ecflow.Task("t1")

t1.add_late( late )
```

### 6.5 Control Structures and Looping

In Python there is not a switch/case statement, however this can be worked round using nested if..elif..else comands.

```
var = "aa"

if var in ( "a", "aa", "aaa" ) : print "it is a kind of a ";

elif var in ( "b", "bb", "bb" ) :print "it is a kind of b ";

else :                               print "it is something else ";
```

#### 6.5.1 Using for loops

```
suite = ecflow.Suite("x")

previous = 0

for i in (0,6,12,18,24) :

    fam = suite.add_family(str(i))

    if i != 0 :

        fam.add_trigger("./" + previous + " == complete ")

    fam.add_task("t1")

    fam.add_task("t2").add_trigger("t1 == complete")

    previous = str(i)
```

### 6.6 Adding externs automatically

Extern refers to nodes that have not yet been defined typically due to cross suite dependencies. Trigger and complete expressions may refer to paths and variables in other suites that have not been loaded yet. The references to node paths and variable must exist, or exist as externs. Externs can be added manually or automatically

Manual Method;

```
defs = ecflow.Defs("file.def") # open and load file 'file.def' into memory
defs.add_extern("/temp/bill:event_name")
```

Automatic Method; this will scan all trigger and complete expressions, looking for paths and variables that have not been defined. The added benefit of this approach is that duplicates will not be added. It is the user's responsibility to check that extern's are eventually defined otherwise trigger expression will not evaluate correctly

```
defs = ecflow.Defs("file.def") # open and load file 'file.def' into memory
.....
defs.auto_add_extern(True) # True means remove existing extern first.
```

## 6.7 Checking the suite definition

The following python code shows how to check expression and limits.

Checking existing definition file that has been saved as a file;

```
def = ecflow.Defs("/my/path.def") # will load file '/my/path.def' from disk
print def.check() # check trigger expressions and limits
```

Here is another example where we create the suite definition on the fly. In fact using the python API allows for a correct by construction paradigm.

```
defs = ecflow.Defs() # create a empty defs
suite = defs.add_suite("s1"); # create a suite 's1' and add to defs
task = suite.add_task("t1"); # create a task 't1' and add to suite
task.add_trigger("t2 == active") # mismatched brackets
result = defs.check(); # check trigger expressions and limits
print "Message: " + result + ""
assert len(result) != 0, "Expected Error: mis-matched brackets in expression."
```

## 6.8 Checking Job Generation

Job generation involves the following processes:

- Locating the '.ecf' files
- Locating the includes files specified in the '.ecf' file
- Removing comment and manual pre-processor statements
- Variable substitution
- Creating the job file on disk

This process can be checked on the client side, with the python API. Since this API is used for checking, the jobs are all generated with the extension '.job0'. The following example checks job generation for all tasks.

```
defs = ecflow.Defs('my.def')          # load file 'my.def' into memory
job_ctrl = ecflow.JobGenCtrl()
defs.check_job_generation(job_ctrl)    # job files generated to ECF_JOB
print job_ctrl.get_error_msg()        # report any errors in job generation
```

For brevity the following examples do not show how the 'defs' object. This can be read in from disk as shown above or created directly in python. This example shows checking of job generation for all tasks under '/suite/to\_check'

```
job_ctrl = ecflow.JobGenCtrl()
job_ctrl.set_node_path('/suite/to_check') # hierarchical job generation under /suite/to_check
defs.check_job_generation(job_ctrl)      # do the check
print job_ctrl.get_error_msg()          # report any errors in job generation
```

This example shows checking of job generation for all tasks, but where the jobs are generated to a user specified directory. i.e. '/tmp/ECF\_NAME.job0'

```
job_ctrl = ecflow.JobGenCtrl()
job_ctrl.set_dir_for_job_generation("/tmp") # generate jobs file under this directory
defs.check_job_generation(job_ctrl)
print job_ctrl.get_error_msg()
```

This example show job checking to an automatically generated temporary directory \$TMPDIR/ecf\_check\_job\_generation/ECF\_NAME.job0

```
job_ctrl = ecflow.JobGenCtrl()
job_ctrl.generate_temp_dir()
defs.check_job_generation(job_ctrl)
print job_ctrl.get_error_msg()
```

## 6.9 Handling Dummy Tasks

Sometimes tasks are created for which there is no associated '.ecf' file. During job generation checking via the python API, these tasks will show as errors. To suppress job generation errors, a task can be marked as a dummy task.

```
the_task = ecflow.Task()
the_task.add_variable("ECF_DUMMY_TASK", "any")
```

## 6.10 Simulation of a running suite

The python API allows simulation. Simulation has the following benefits:

- Exercise the suite definition. There is no need for '.ecf' files
- Can be done on the client side, no need for server
- Can help in detecting deadlock's
- Will simulate with both 'real' and 'hybrid' clocks
- A year's simulation can be done in a few minutes. Small definitions can be simulated in a few seconds

There are however restrictions. If the definition has large loops due to Repeat date attributes, which run indefinitely, then in this case the simulation will never complete, and will timeout after a years worth of run time. Hence it's best to restrict simulation, to definitions which are known to complete.

If the simulation does not complete it will produce two files, which will help in the analysis:

- defs.depth: This file shows a depth first view, of why simulation did not complete.
- defs.flat: This shows a simple flat view, of why simulation did not complete

Both files will show which nodes are holding, and include the state of the holding trigger expressions.

```
def simulate_deadlock():  
  
    # This simulation is expected to fail, since we have a deadlock/ race condition  
  
    defs = ecf.Defcs()          # create a empty defs  
  
    suite = defs.add_suite("dead_lock")  
  
    fam = suite.add_family("family")  
  
    fam.add_task("t1").add_trigger("t2 == complete")  
  
    fam.add_task("t2").add_trigger("t1 == complete")  
  
    theResult = defs.simulate(); # simulate the definition  
  
    assert len(theResult) != 0, "Expected simulation to return errors"  
  
    print theResult  
  
if __name__ == "__main__":  
  
    simulate_deadlock()
```

## 6.11 Error Handling

Any errors in the creation of suite are handled by throwing an exception.

```
try :  
    defs = ecflow.Defs()      # create a empty definition  
    s1 = defs.add_suite("s1") # create a suite "s1" and add to defs  
    s2 = defs.add_suite("s1") # Exception thrown trying to add suite "s1" again  
except RuntimeError, e :  
    print e
```

## 7 The ecFlow Server

### 7.1 Starting the ecFlow Server

The command `ecflow_server` is used to start an `ecflow_server` using the default port number or that defined by the variable `ECF_PORT`.

```
cd ECF_dir1
ecflow_server & # start ecFlow with default port 3141
```

Multiple ECFs can be run on the same host using different port numbers. There are two mechanisms for specify the port number:

- Using arguments on the command line. i.e. `ecflow_server --port=3141`
- Using Environment variable. `ECF_PORT`

If both are specified the command line argument takes precedence

```
cd ../ECF_dir2
ecflow_server --port=3142 & # start ecFlow with port number 3142
cd ../ECF_dir3
export ECF_PORT=3143
ecflow_server & # starts ecFlow with port number of 3143
```

**Note:** the ECFs are started in different directories so that the output and checkpoint files are not overwritten

Adding a new server to `ecflowview` adds the definition to the file `~/ecflowview/servers`. This can be modified directly.

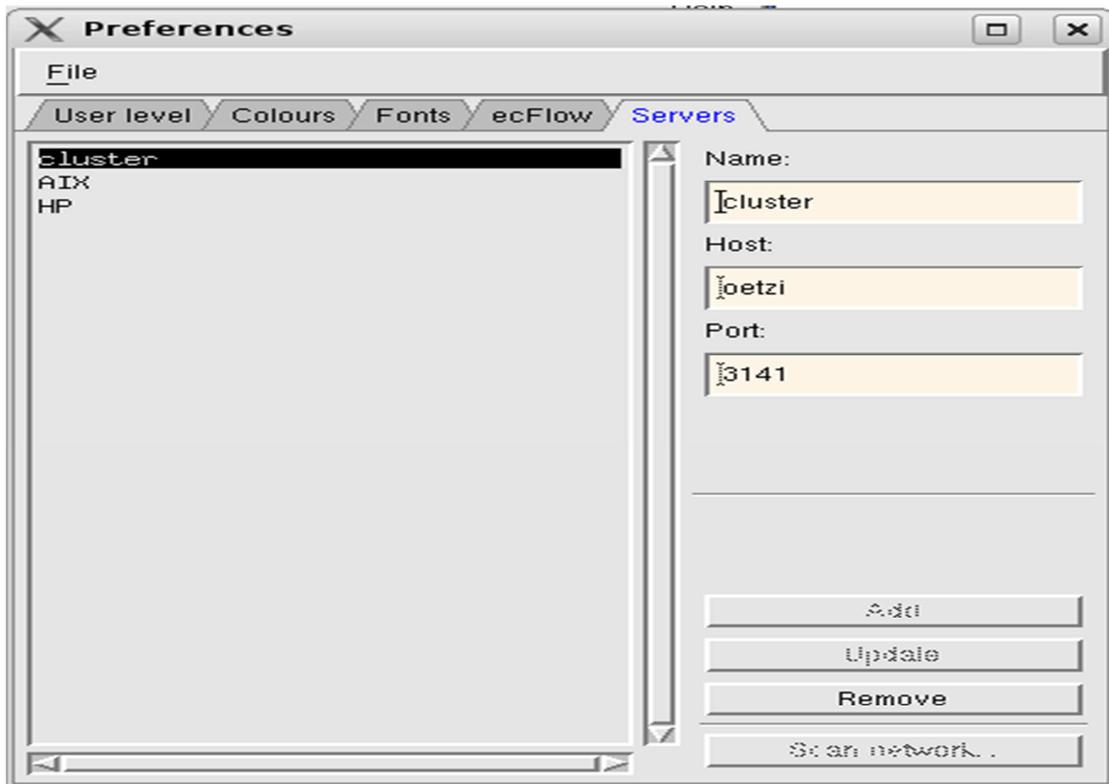
You cannot start two `ecFlow` servers on the same machine with the same port number. To simplify users wanting their own `ecFlow` servers we have a script `ecflow_start.sh` (an example is included in the latest releases of ECF) that sets up an `ecFlow` server using a port number based on the users own unique user ID

You can check what port numbers are being used, with `netstat`: To list all open network ports on your machine, run **`netstat -lnptu`**.

Here is a breakdown of the parameters:

- `l` - List all listening ports
- `n` - Display the numeric IP addresses (i.e., don't do reverse DNS lookups)
- `p` - List the process name that is attached to that port
- `t` - List all TCP connections
- `u` - List all UDP connections

Figure 7-1 Viewing new ECFLOW servers with ecfview.



When using non-default ecFlow servers, ecfview needs to be configured to recognise the port used. Opening the edit-preferences under ecfview and selecting the servers tab you can describe your new ecFlow server to ecfview.

Adding a new server to ecfview adds the definition to the file `~/xecfrc/servers`. This can be modified directly.

It is a good idea to use a start up script that automatically sets up a unique port number.

## 7.2 Stopping ecFlow servers

To safely stop an ECFLOW server you should “halt”, “check point” and then “terminate” the server. This can be done either through ecfview (right click on the server) or directly by the CLI commands “halt”, “check\_pt” and “terminate”.

```
# halt server, write out the in memory definition as a check  
# point file, then terminate the server  
ecflow_client --group="halt; check_pt; terminate"
```

## 7.3 Checking if an ECFLOW server is running on a host

You can check if an ECFLOW server is running on a system using the ping command, e.g.

```

# Check if server is running on 'localhost' on port 3141
ecflow_client --ping

# Check if server running on machine 'fred'
ecflow_client --ping --host=fred

# Check if server running on 'fred' with port 3222
ecflow_client --ping --host=fred --port=3222

# Check if server running using ECF_PORT and ECF_NODE
export ECF_PORT=3144
export ECF_NODE=fred
ecflow_client --ping

```

Note: when ECF\_NODE and ECF\_PORT are used in conjunction with command line arguments, then the command line argument take precedence.

## 7.4 Start-up files for ECFLOW server

When ecFlow is started up it uses a number of files for configuration and reporting. These files can be configured through environment variables or in an ecFlow configuration file (e.g. the file `server_environment.config` in the ecFlow source directories).

The first thing ecFlow does is to change the *current working directory* (or CWD) into `ECF_HOME`, so all the other files listed there are read from that directory (unless full path names are used). This directory must be accessible and writable by the user starting ecFlow otherwise ecFlow cannot start.

### 7.4.1 ECFLOW log file

If this file exists at the beginning of ecFlow execution, it is appended to. Otherwise it is created. The location and name of the log file can be configured using the environment variable `ECF_LOG`, which can be set before the ecFlow server starts.

Symbol	Meaning
MSG	Information message generated by user action, normal operation.
LOG	Information message generated by task or ECF, normal operation. These are mostly messages generated when nodes go through status changes.
ERR	An error message, abnormal operation the action could not be done Some errors are ignored on client side, for example trying to send a label or an event that does not exist in ecFlow is an error on the ecFlow side but not on client side.

WAR	Warning message, not an error but corrective action was taken. For example using an old version of client may cause a warning message to be printed.
DBG	Debugging message, by default these are not visible they must be turned on by a privileged user.
others	There could be a keyboard echo in the log file if ecFlow is running interactively; these lines should be ignored by programs processing the log file.

The log file contains all the actions of the ecFlow server and shows information such as the halt, shutdown etc. It is a pure text file and can easily be processed by other programs. For example ecfview uses this file to show the *timeline* window.

The syntax of the log file is fairly simple. For each action in ecFlow a line is output. The format is as follows:

```
XXX:[HH:MM:SS D.M.YYYY] command:fullname [+additional information]
```

Where XXX is one of the symbols in the above table. The *timestamp* inside [] is the system time, not the suite time, which may differ.

**Note:** that user commands often generate both **MSG** and **LOG** level messages. **MSG** is the command executed and **LOG** is the effect it had on the node or nodes.

**Note:** The log file is not removed so it needs to be managed. We tend to compress and archive our log files each day using an ecFlow controlled script.

## 7.4.2 ECFLOW check point file and failure tolerance

There are a number of reasons that could cause ECFLOW to stop working (server crashes, the computer ECFLOW is running on crashes, etc.). The ecFlow checkpoint file allows ecFlow to restart at the point of the last checkpoint before a failure. This gives reasonable tolerance against failures.

When the server starts, if the checkpoint file exists and is readable and is complete, ECFLOW server recovers from that file. Once recovered the status of server may not exactly reflect the real status of the suite, it could be up to a few minutes old. Tasks that were running may have now completed so the task status should be checked for consistency.

The checkpoint files can be read by any ecFlow running on any operating system

There are two separate checkpoint files.

ECFCHECK

ecf.check

ECFCHECKOLD

ecf.check.b

When ecFlow needs to write a checkpoint file it first moves (renames) the previous file ECF\_CHECK to ECF\_CHECKOLD and then creates a new file with the name ECF\_CHECK. This means that you should always have a file that is good. If a crash happens while writing ECF\_CHECK, you can still recover from ECF\_CHECKOLD (by copying its contents to ECF\_CHECK), although that version is not quite as up to date.

Tip: You can copy the checkpoint files between systems. Another ecFlow server can be started with the original server's checkpoint file and take over from the original ecFlow server host in case of a catastrophic systems failure.

## 7.5 Security

An ecFlow server is started by one user account and all tasks are submitted by this user account by default. The advantage of this open way of working is that anyone can support your suite, which can of course be the disadvantage. Tasks (or suites) can be run using other user IDs as allowed by standard UNIX permission.

At ECMWF we run ecFlow in a relatively open way. We have decided to limit the number of accounts/users running ecFlow to simplify cooperation and file permission problems. Most research ecFlow servers run under one research account allowing greater cooperation. However, for operations we want to limit full access to a handful of trusted users, whilst giving others the ability to monitor the operational suites. We use the ecFlow white list file to limit access in operations.

## 7.6 Security: ECFLOW White list file

ECFLOW white list file is a way of restricting the access to ecFlow to only known users.

The file lists users with full access and users with only read access. The read-only user names start with '-' (dash/minus). Note you must include a version number, e.g.

The environment variable ECF\_LISTS is used to point to the white list file.

The white list file is an ASCII file.

```
File ecf.lists
#
4.4.14 # whitelist version number
#
# Maintenance group and operators
#
uid1
```

```
uid2
cog
#
# Read-only users
#
-uid3
-uid4
```

If you edit this file while ecFlow is running you need to use the following command to activate the change in ECF:

```
ecflow_client --reloadwsfile
```

## 7.7 Handling Output

### 7.7.1 ECFLOW stderr and stdout

Normally you run the ECFLOW server in the *background* and the **stderr** and **stdout** is redirected to **/dev/null** by executing the following command in a start-up script:

```
ecflow_server > /dev/null 1>&2
```

When learning how to use ECF, you may open a window and run ecFlow in that window interactively. Notice that server still writes the log file (into **ECF\_LOG**.)

### 7.7.2 ecFlow log server

The default behaviour of the ecflowview client is to access the output file directly (case 1 in Figure 7-2).

When this is not possible, e.g. when the ecflowview host cannot see the relevant file system, the ecFlow server is asked to request the output (case 2 in Figure 7-2). If the output file is large the ecFlow server will supply the last 10000 lines of the output. You can use the following command to get the relevant file associated with a given node:

```
ecflow_client --file=node_path [script, | job | jobout | manual | stat]
```

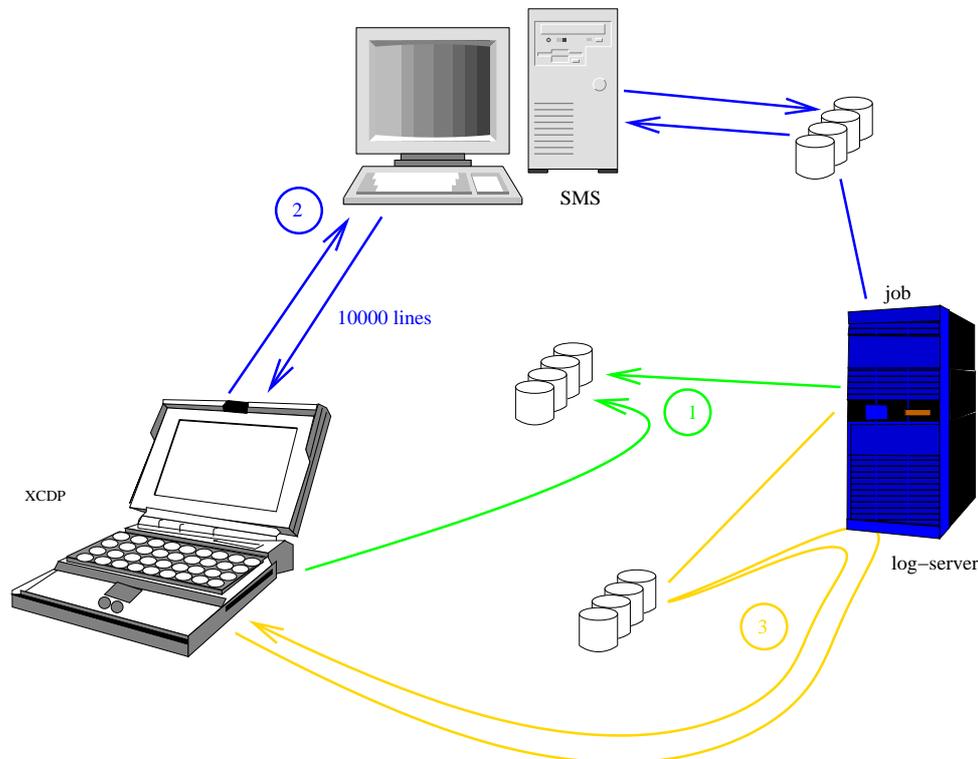
This will output the file to standard output. This capability uses ecFlow to get the file. The original file can be located in a directory that is visible to the ECF, but not to the client.

To view output from a server where the ecFlow server does not have access to the file systems we can use a log server (case 3 in Figure 7-2).

Using ecflowview, you click on the manual, script, output or job buttons. If you configured your ecflowview to retrieve files locally (Edit/preferences/SMS) ecflowview first looks if the required file is directly accessible. If not, it looks into the suite definition for the variables **ECF\_LOGHOST** and **ECF\_LOGPORT** to retrieve the file from the ecFlow log server. If the log server does not respond it contacts the ecFlow server and retrieves the file from it. This

consumes available resources for the ecFlow server, so the log server is useful in reducing ecFlow server load when many users try to get large output files.

Figure 7-2. Accessing job output, using ecflowview.



The log server consists in a Perl script `logsvr.pl` launched by a shell script. These are available in the ecFlow distribution. The log server uses three variables set by the shell script:

- `ECF_LOGHOST`: the name of the log server host
- `ECF_LOGPORT` : the port used by the log server (typically we use 9316)
- `ECF_LOGPATH` : main path where the scripts may be found below, on the target host  
e.g. `export ECF_LOGPATH=path1:path2:path3`
- `ECF_LOGMAP` : indicates the log server how to transform the file name to retrieve the file locally,  
e.g. `export ECFLOGMAP=path1:path1:path1:local1:path2:path2:path2:local2` This indicates that a mapping between `local1` by `path1` from the file name when name fits, or `local2` by `path2` ..., `path1` is local to the target host, and `local1` is local to the ecFlow server. The log server is launched on the target host (`ECFLOGHOST`).

You can contact the log server manually using

```
telnet <host> <port> get_file_path
```

## 8 ECFLOW CLI (Command Level Interface)

The command level interface is provided by the `'ecflow_client'` executable. By calling:

```
ecflow_client --help
```

We can get the list of all the available commands.

Most users of ecFlow will be happy to use ecflowview, but some things cannot be done using ecflowview. Note also that most of the commands that you execute using ecflowview are actually CLI commands and that you can execute CLI commands in ecflowview's *collector window*.

CLI is *command line based*, which means it reads your commands a line at the time.

The very first argument to `'ecflow_client'` must begin with `'--'`

```
ecflow_client --load=/my/home/fred.def
```

For further information please see the ecFlow reference manual.

### 8.1 get

The CLI command `get` can be used to *show* the definition that is loaded in the server.

Get all suite node trees from the server and write to standard out.

```
ecflow_client --get
```

This gets the suite "s1" from the server, and writes to standard out. In both of the examples the output is fully parse-able

```
ecflow_client --get=/s1
```

To write the node tree *\*state\** to standard output please use group option, i.e.

```
ecflow_client --get_state
```

```
ecflow_client --get_state= /s1
```

This output from 'show state' is not parse-able

### 8.2 CLI scripting in batch

You can use the CLI from within your tasks. This gives you some very powerful tools for controlling your suite and can even allow you to set up dynamic suites.

You can alter ecFlow variables (using alter), set particular tasks or families complete (using force) and even generate dynamic suites. To do this you could modify a definition file template and replace the modified part of the suite.

## 8.3 Configuring ECFLOW

### 8.3.1 server\_environment.cfg

The file server\_environment.cfg can be found in the ecflow/Server folder. Modifying this allows you to change the defaults environment variables for ecFlow including the default commands for submitting and killing tasks. The server look for the file in the CWD, hence make sure you move it.

Shows some of the default variables you can configure.

Table 8-1 Some configurable variables in config.h

Variable	Description
ECF_LOG	Name of the ecFlow log file
ECF_OUT	(stdout and stderr of the process ECF)
ECF_CHECK	Name of the ecFlow checkpoint file
ECF_CHECKOLD	Name of the ecFlow backup checkpoint file
ECF_INTERVAL	The frequency of checking time dependencies
ECF_CHECKINTERVAL	Default time between automatic “check pointing”
ECF_JOB_CMD	Default ecFlow submission method
ECF_KILL_CMD	Default ecFlow kill method

### 8.3.2 Start-up scripts

Start-up scripts are another useful way of starting and configuring ECF. There is an example included in the default installation of ECF; ecflow\_start.sh.

Scripts like these can be used to reconfigure some of the default ecFlow variables and check that ecFlow is not already running (using ecflow\_client --ping), set the environment (or do it in .profile or .cshrc), backup logfiles and checkpoint files, start ecflow\_server in the background and alert operators if there is a problem starting.

An automatic start-up script is typically located in /etc/rc2.d/? with su - “normal-user”.

## 8.4 Compiler and OS requirements

ECFLOW is available for UNIX systems only.

Table 8-2 shows on the operation systems and compiler requirement to build ECF. Note some systems only run the client part and ecfview is not available.

Table 8-2 Operating systems on which ECFLOW has run

Make	Compiler	Operating system and version
HP	aCC: HP C/aC++ B3910B A.06.20	HP/UX 11.23
IBM	c++/vacpp/11.1.0.1	AIX Version 5.3
LINUX	gcc 4.2.1, 4.5	SuSe 10.3
Linux Cluster 64 bit	gcc 4.2.1, 4.5	Suse 10.3,11.3

Highlighted systems are actively tested at ECMWF.

## 9 Flags used by ECFLOW

Each node in ecFlow may have a set of flags *set*. Most of them are just informative flags, but some may control the running of the node.

Most of the flags are *cleared* when the suite is begun. And some flags have other information associated with them, such as *message* which raises a flag. Table 9-1 Flags used by ecFlow shows the flags used by ECF.

Table 9-1 Flags used by ecFlow

Name of the flag	Symbol used by ecflowview if any	Description and limitations
force-aborted		node has been forced to aborted status
user-edit		user edit failed (only for tasks)
task-aborted		the running of the job failed (it finished with <b>abort</b> )
edit-failed		ecFlow pre-processor editing failed (the .job file cannot be created)
ecfcmd-failed		job could not be submitted ( <b>ECF_JOB_CMD</b> failed)
no-script		ecFlow could not find the script
Killed		killed by user (only tasks and aliases)
migrated		node has been migrated (suites and families)
late		node is late
message		has user messages, user has issued a command since last begin
By_rule		node was forced complete by rule
queuelimit		queue limit reached
task-waiting		running task is waiting for trigger, task is active but is waiting

Name of the flag	Symbol used by ecfview if any	Description and limitations
		for something in ecFlow
locked		node is locked by a user

ecflowview also shows some *pseudo flags* which are just markers that something else is available, like if a node has a *time dependency* ecfview may draw a symbol next to the node to indicate it.

Table 9-2 Pseudo flags used by ecfview

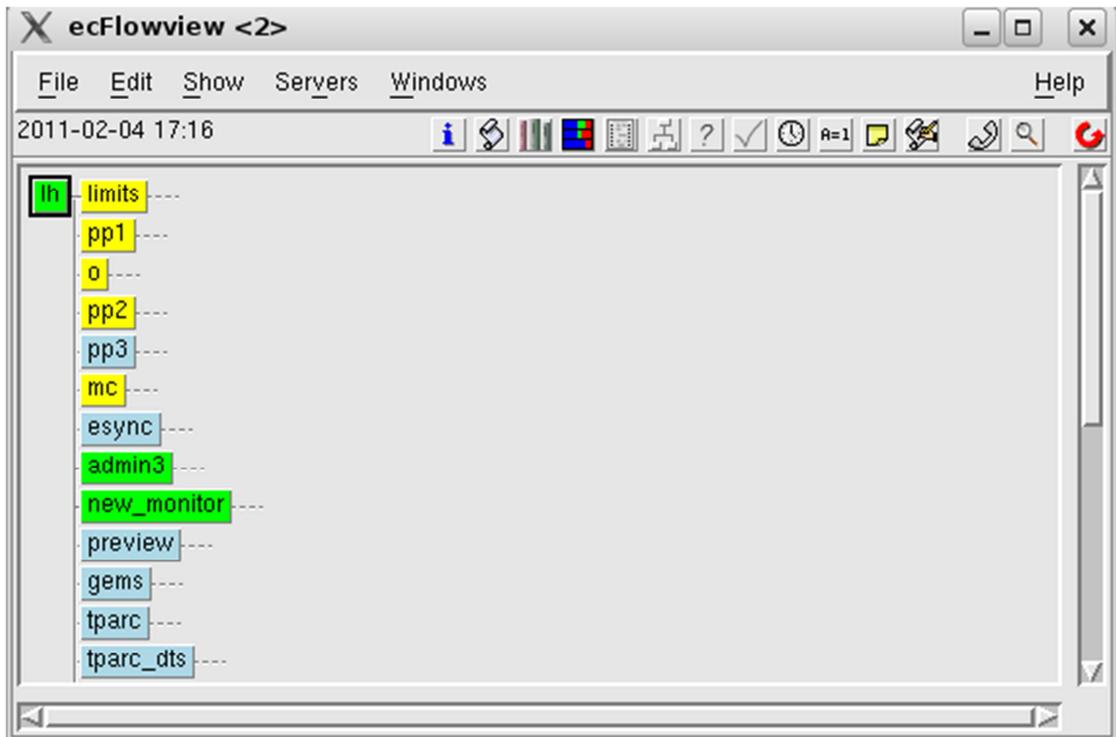
Name of the flag	Symbol used by ecfview	Description and limitations
Time dependency		node has got a “time” dependency
Date dependency		node has got a “date” or “day” or dependency

## 10 ecfview

ecflowview is the GUI based client. It is an XWindow/Motif based application that displays graphically the status of the tasks controlled by a supervisor. ecfview displays the hierarchy suite/family/task in a tree fashion, using colour coding to reflect the status of each node. Every attribute can be shown in the tree window.

The application is started by typing “**ecflowview**” at a UNIX prompt. Once the program is running you can select the required server from the Servers menu. You should get a display similar to the following:

Figure 10-1 ecfview window



The boxes represent nodes. ecfview uses the three buttons of the mouse to perform different actions. The following figure show how the mouse buttons are used in ecfview:

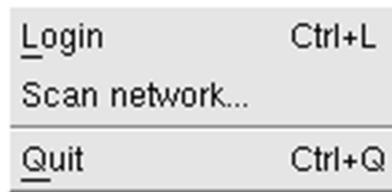
Figure 10-2 Mouse usage in ecfview



## 10.1 Main window menus

### 10.1.1 File

Figure 10-3 Main window file tag



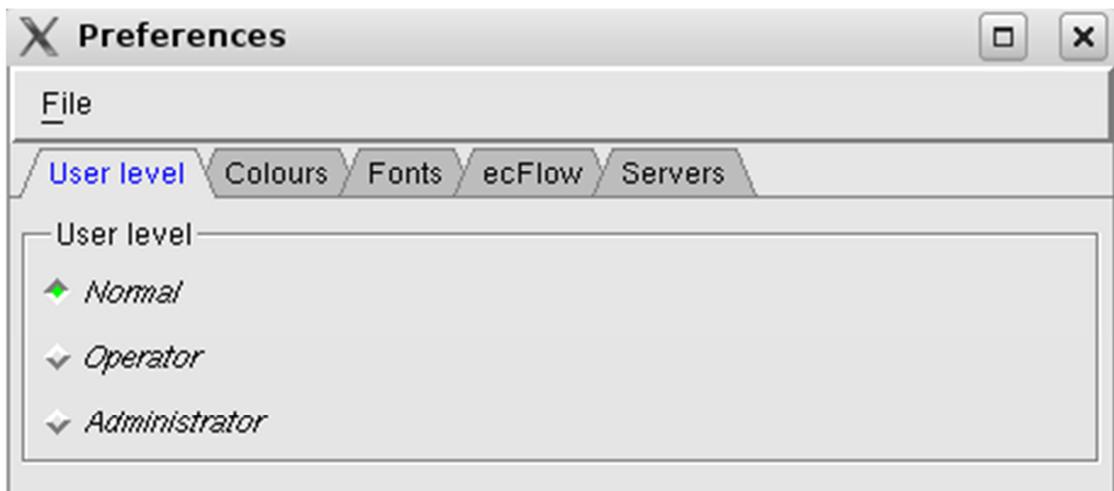
- Login – this allows you to log in to a specific ECFLOW server but only if it using the default port number.
- Scan network – scans network for all ecFlow servers – should be used with caution as will check every server it can find and may take some time to complete.
- Quit – exits the ecfview application.

Note: Some menu items in ecfview have a tear off strip, which opens the menu in a new window.

### 10.1.2 Edit

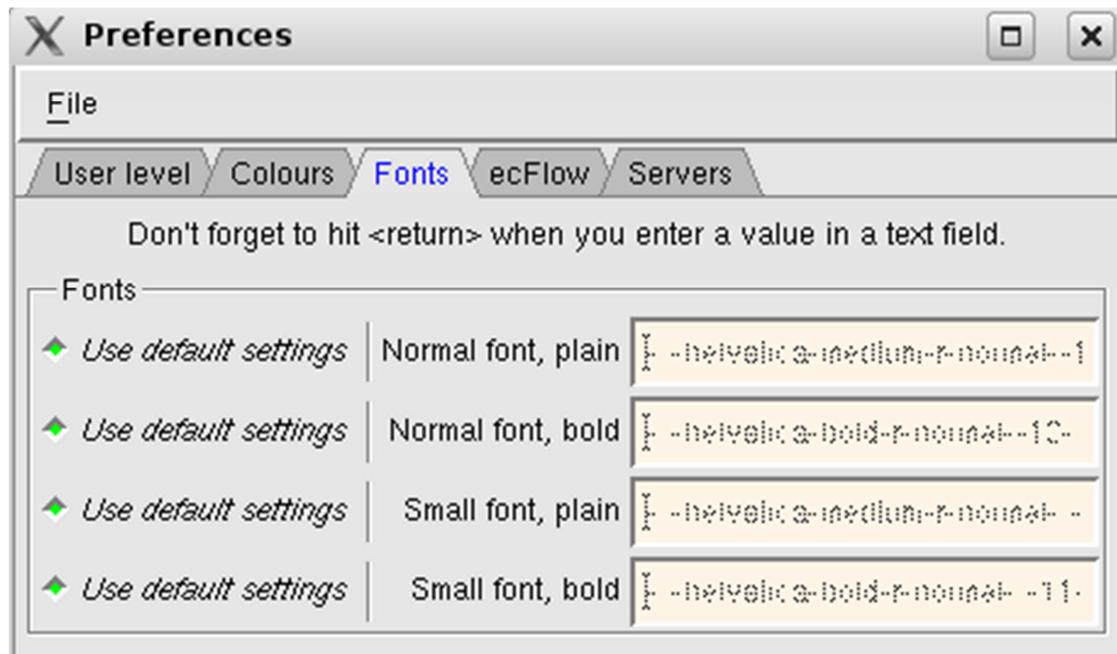
The edit tag has one available option – Preferences. Through this you can set a number of options in ecfview.

Figure 10-4 Main window edit menu->preferences – user level folder



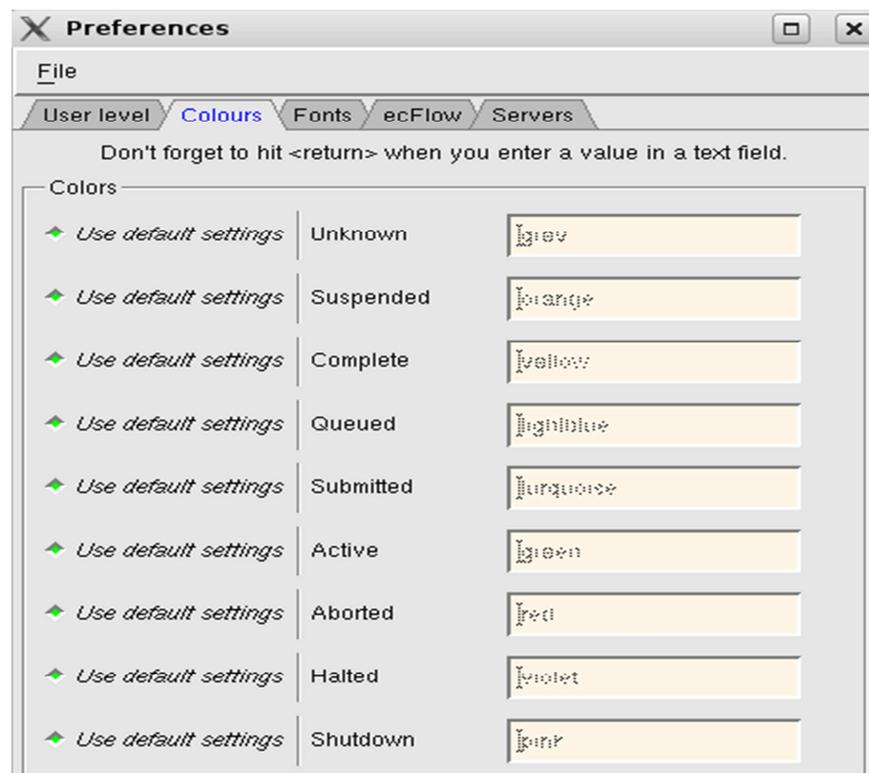
The user level tag in the preferences section of the edit tag allows you to define the “User level” for your ecfview access. The different levels modify the available menu options in ecfview. However, the behaviour can be overwritten or modified by the contents of a local .ecflowrc file.

Figure 10-5 Main window edit menu→preferences – fonts folder



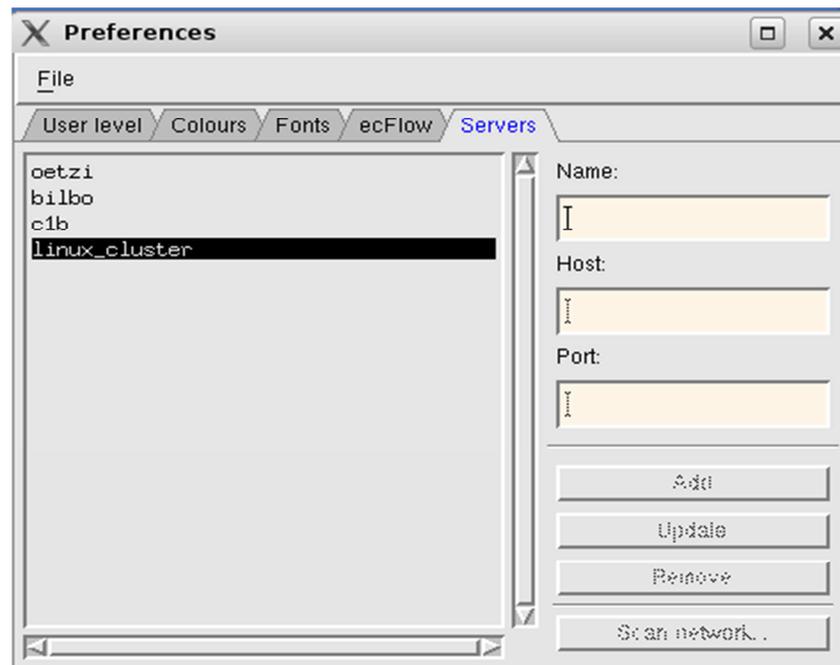
The font's folder allows redefinition of the fonts used in ecflowview.

Figure 10-6 Main window edit menu→preferences – colours tag



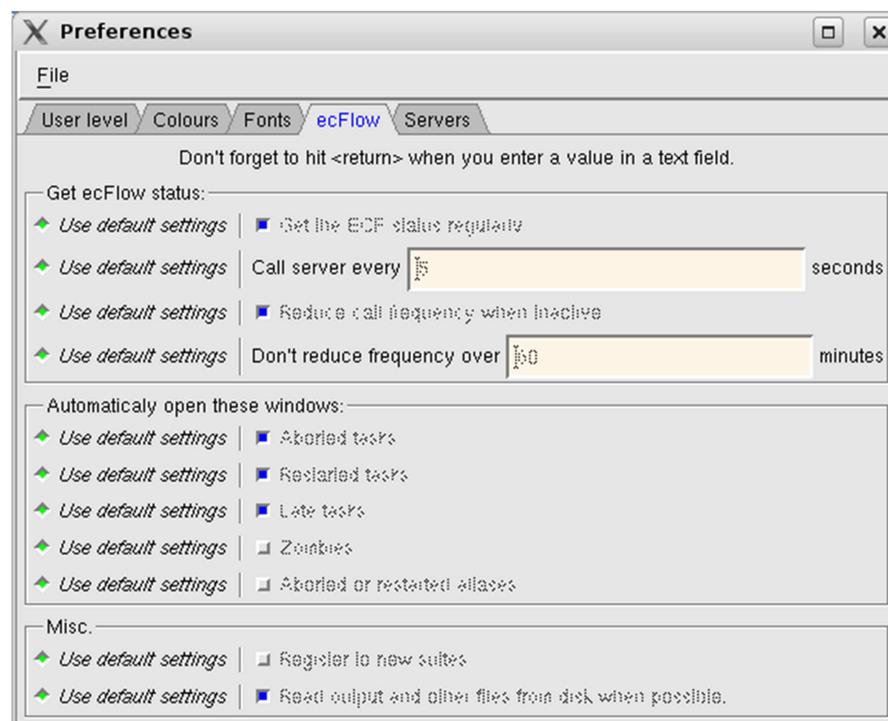
The colours tag allows you to redefine the default colours used by ecflowview. This is useful if you have difficulty distinguishing certain colours.

Figure 10-7 Main window edit menu-> preferences – servers folder



The server's folder allows you to define additional ecFlow servers not available to the global list. To add a new entry you define the name you wish to use for the ecFlow server together with the host name and port number. These servers will then be added to your local .ecflowrc file and be available from the main servers tag in ecflowview.

Figure 10-8 Main window edit tag – preferences – ecFlow folder

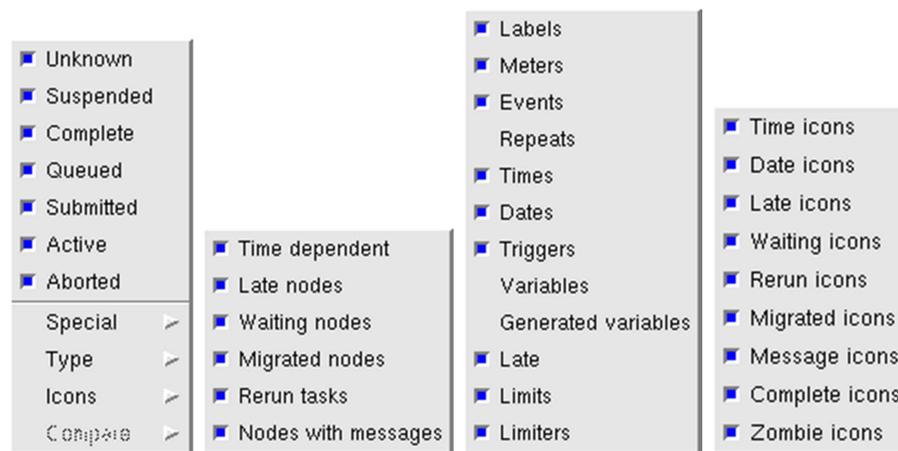


The final folder allows you to define more ecflowview settings including the frequency ecflowview polls the servers, when to open new windows, whether you wish to automatically register to new suites on ecFlow servers and whether you want the output to be automatically updated or not.

### 10.1.3 Show

This tag is used to define what ecflowview will nodes, icons or type's ecflowview will display. This way ecflowview is used, varies from user to user. A developer may be interested in seeing all details regarding tasks in an ecFlow tree, whilst an operator may only be interested in seeing tasks that are submitted, active or aborted.

Figure 10-9 Main window show tag – main, special, type and icons menus



### 10.1.4 Servers

This folder displays all the ECFLOW servers either defined in the global ecflowview configuration or defined locally in your .ecflowrc file. Clicking on the server name toggles the visibility of the ecFlow server.

Figure 10-10 Main window servers folder

<input checked="" type="checkbox"/> localhost	hallas	xpr
<input checked="" type="checkbox"/> od	hasms	xpt
<input type="checkbox"/> od-backup	naw	xse
<input type="checkbox"/> haod	smszum	xtr
<input checked="" type="checkbox"/> od2	systems	zfi
<input type="checkbox"/> od2-backup	vingolf	mac
<input type="checkbox"/> haod2	w3app	kelvyn
<input checked="" type="checkbox"/> od3	xat	bilbo
<input type="checkbox"/> od4	xbe	test
<input type="checkbox"/> ode	xch	ugm

## 10.2 ecfview “buttons”

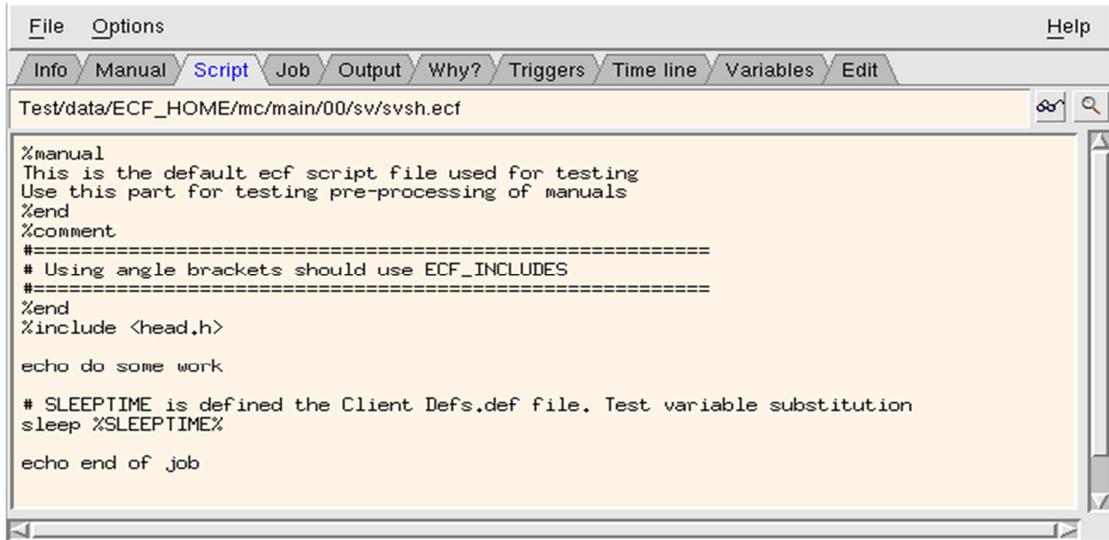
On the top right hand side of the main ecfview window (see Figure 10-1) there are a number of buttons. Table 10-1 gives a description of each button. Holding the mouse pointer over the buttons on ecfview shows the function of each button.

Table 10-1 ecfview buttons

Button	Usage
	Info on definition of node c.f. entry in definition file.
	<a href="#">View the script</a> (see Figure 10-11). This is only available for tasks.
	View the manual pages for the node.
	View the job status. This invokes the ECF_STATUSCMD for running or submitted jobs,
	View the job output. This is only available for tasks.
	View the node triggers.
	“Why?” Gives information on why a queued node has not been submitted. A job may not be scheduled by ECFLOW for various reasons. ECFLOW will produce a report by scanning the triggers, the date and time dependencies, the limits and the status of various nodes.
	Display time-line for node. By reading ecFlow log files, ecfview can build a time map of the various events recorded in the log files (starting jobs, completing jobs, distance between two events...). These events can be sorted in various ways. For each task, ecfview can give a summary of its activity. It has already proved to be very useful in spotting some delays in the archive jobs due to poor data transfer rates. It has also been used to organise the frequency of acquisitions jobs within an hour. Interesting queries can be answered, such as “what jobs are running at the same time as the forecast?”. In conjunction with the dependency information, this will help to improve the design of the operational suite.
	Display and modify variables for given node.
	Display messages relevant to node.
	Edit the particular task and submit with options for alias and pre-processing.
	Chat - an internal mailing system. This is an obsolete feature.

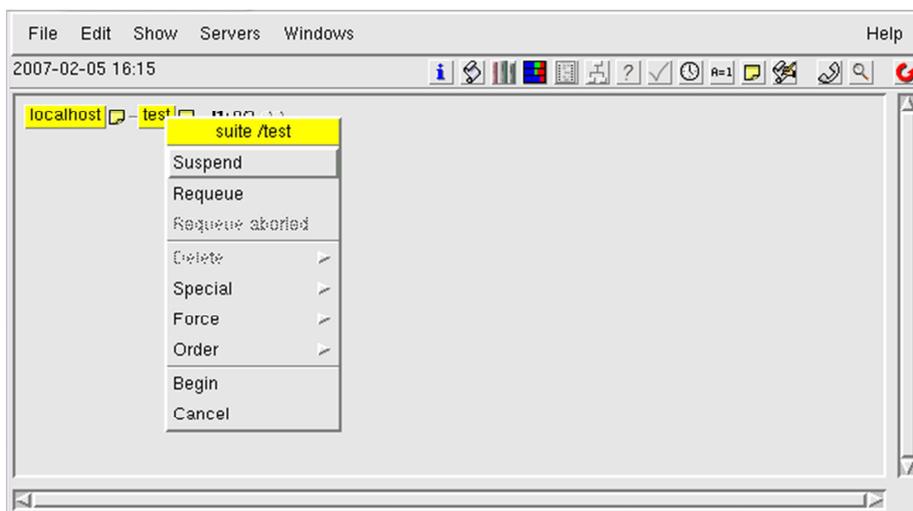
Button	Usage
	Search for particular tasks, variables etc.
	Refreshes the current display.

Figure 10-11 Displaying the script for a task



To issue a command to a suite, such as run the suite, move the mouse pointer over test and click on the right mouse button. A pop-up menu will appear. Choose begin. (If the menu does not offer a begin option, select Preferences... in the Edit menu and set the User level to Administrator)

Figure 10-12 Pop up menu for a suite



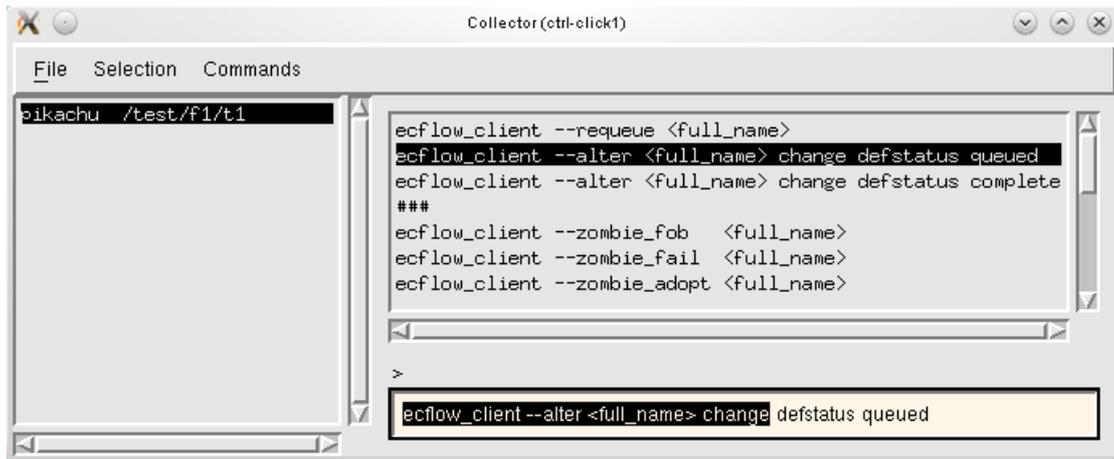
### 10.3 Menus

To perform any actions on a node, such as suspending a task or editing a label, right click on the node. This will bring up a menu dependant on the type of node selected. The menu is built up from a file that describes what commands can be performed on what node. Through these menus the user has access to a range of CLI commands.

### 10.4 The Collector

A very common wish from ecflowview users is the ability to issue a command on several nodes, such as resuming all suspended tasks at once. By holding down the ctrl key whilst left-clicking on nodes you can use the collector or by collecting the result of the search command. These appear in the left hand box seen in Figure 10-13. The user can then type in a CLI command in the bottom right box that will be executed for each node of the selection. As in the command menu, the user has access to the complete range of available CDP commands. The tag `<full_name>` represents all the selected entries in the left hand box.

Figure 10-13 ecflowview Collector



Collector commands include:

`ecflow_client --alter <full_name> change defstatus (unknown/complete/queued)`

- modify default status of node (unknown/complete/queued)

`ecflow_client --alter <full_name> change variable var value`

- change node variable "var" to value

`ecflow_client --force complete <full_name>` - force node complete

`ecflow_client --run <full_name>` - execute task

`ecflow_client --requeue <full_name>` - requeue node

`ecflow_client --resume <full_name>` - resume node

## 10.5 Searching

Because of the sheer number of nodes that make the suites we are monitoring, ecfLOWview provides a powerful way of searching nodes. Not only can the user search nodes by name, but also they can search the variables and the triggers for occurrences of a given string. It is possible to find what jobs run on a particular machine. The search facility is linked to the node collector so the user can perform an action on the result of a search (see Figure 10-14 and Figure 10-15).

Figure 10-14 ecfLOWview search window

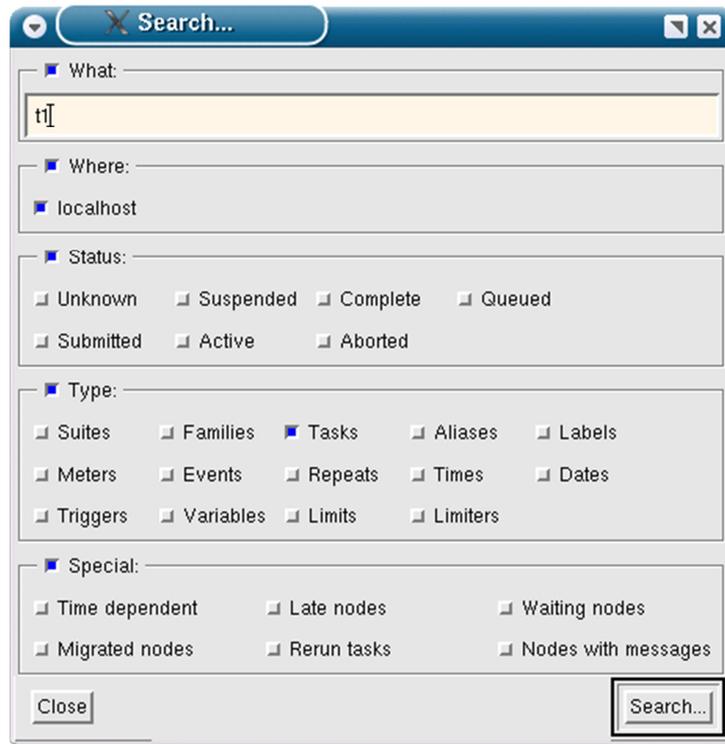
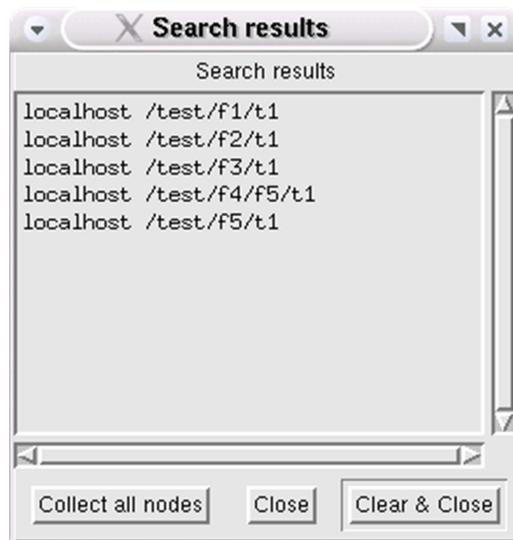


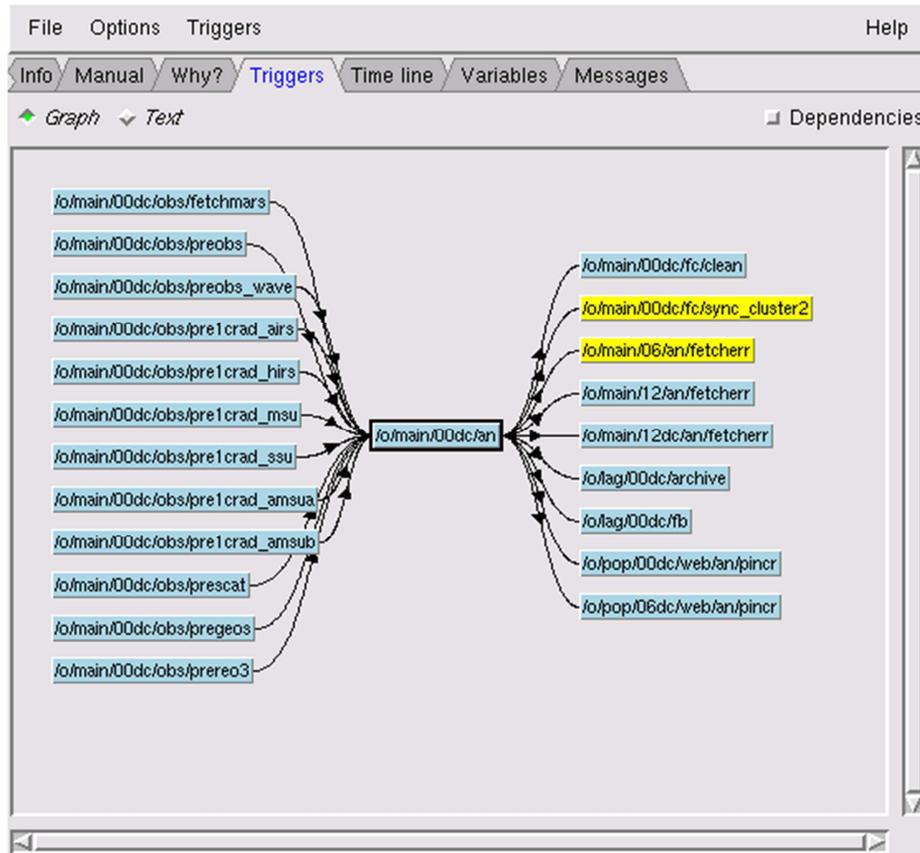
Figure 10-15 ecfLOWview search results window



## 10.6 Dependencies

By defining triggers between task, families and suites, the analyst can create a complex graph of dependencies. ecfLOWview can display this information using Why? as described above or using the triggers button: See Figure 10-16.

Figure 10-16 Triggers



ecfLOWview can display triggers, as they were defined by the analyst, in a hypertext window or in a graph window. The user can browse through the graphs to follow the triggers. Triggers can also be displayed directly in the tree. Figure 10-16 shows some dependencies of an operational analysis. On the right-hand side are the nodes waiting for some data from the operational suite, such as the ensemble forecast (mc) and the limited-area wave model (law). On the left-hand side are the nodes the operational suite is waiting for, in this case some observation handling tasks. Ticking the dependencies on the top right of the window will produce a full set of dependencies for the node.

## 10.7 Editing scripts

Most of the objects visible in ecfLOWview are editable (apart from triggers). Labels, limits, variables, meters, repeat can all be changed by the user. The user can edit the script and the variables of a script. Nevertheless, those changes are only valid for one run of the task; changes that are more permanent must be done directly on the script files.

### 10.7.1 Aliases

A very useful addition to “edit” is the ability to clone a task, perform some minor modifications to its script, and run it as an “alias”. Such a task can run under the control of the ecFlow but has no impact on the activity of the suite itself i.e. it will have no dependencies. This feature can be used to rerun a task for a previous date, or to solve transient problems such as full file systems. In this case, the task can be run using a different disk, in order to guarantee the completion of the suite on schedule. The analyst then has time to understand why this condition arises and take the necessary actions.

### 10.8 Zombies

A zombie is a running job that fails authentication when communicating with the ecflow\_server. As zombie icon will be visible in ecflowview (see Figure 10-17). There are a wide variety of reasons why a zombie is created. The most common causes are due to user action:

- The node tree is deleted, replaced or reloaded whilst jobs are running
- A task is rerun, whilst in a submitted or active state
- A job is forced to new state, i.e. complete

Rarer causes include:

- ecFlow script errors, where we have multiple calls to init and complete child commands
- The child commands in the ecFlow script are placed in the background. In this case order in which the child command contact the server, may be indeterminate.
- Your queuing system might submit a job twice
- the system running your ecFlow server crashes and the recovered check point file is out of date

The default behaviour of the ecFlow server is to block the job. The child command continues attempting to contact the ecFlow server. This is done for period of 24 hours. This period is configurable see ECF\_TIMEOUT on ecflow\_client. The jobs can also configured, so that if the server denies the communication, then the child command can be set to fail immediately. (See ECF\_DENIED on ecflow\_client). ecflowview provides a dialog which lists all the zombies and the actions that can be taken. These include:

- Terminate:  
The child command is asked to fail. Depending on your scripts, this may cause the abort child command to be called, which again will be flagged as a zombie.
- Fob:  
Allow the job to continue. The child command completes and hence no longer blocks the job. Great care should be taken when this action is chosen. If we have two jobs running, they may cause data corruption. Even when we have a single job, issues can arise. i.e. if the associated command was an event child command, then the event would not be set. If this event was used in a trigger expression, it would never evaluate.

- **Delete:**  
Remove the zombie from the server. The job will continue blocking, hence when the child command next contacts the `ecflow_server`, the zombie will re-appear. If the job is killed manually, then this option can be used.
- **Rescue:**  
Adopt the zombie and update the node tree. The `ECF_PASS` on the zombie is copied over to the task, so that the next child command will continue as normal.

Figure 10-17 Zombie icon

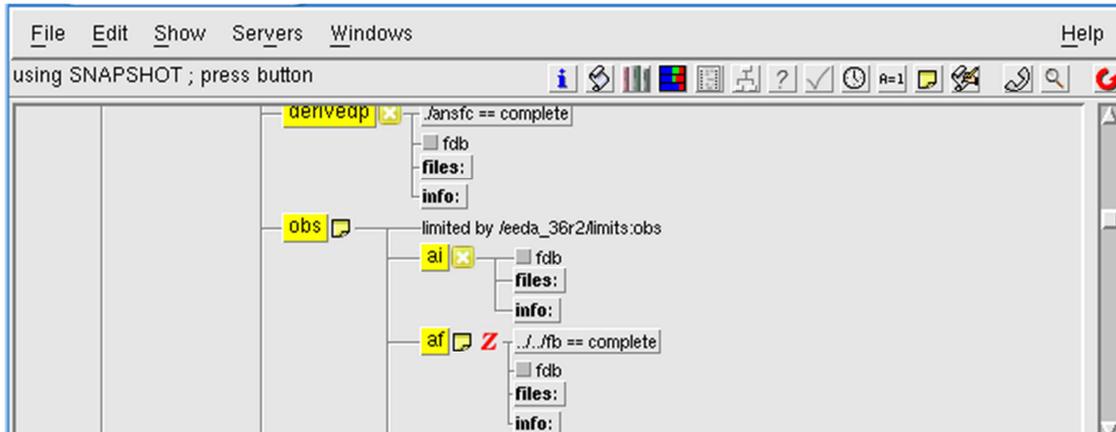
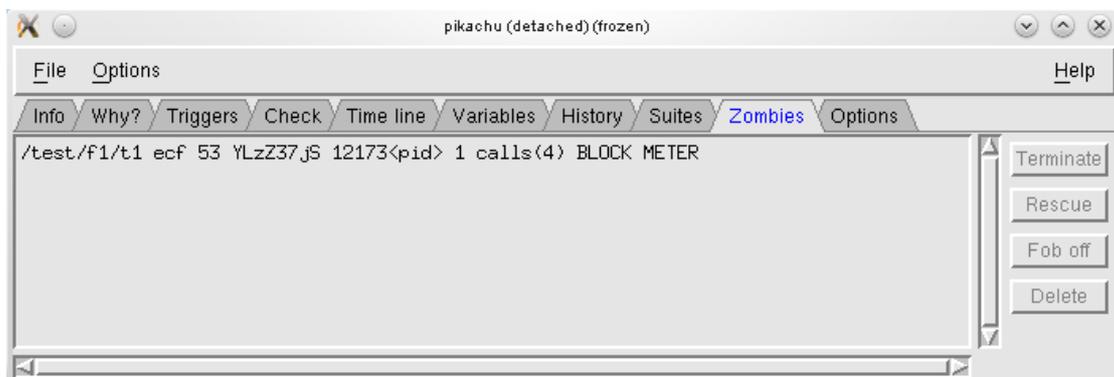


Figure 10-18 Zombie tab available from right clicking on ecFlow server node



## 11 Index

Adding externs automatically .....	67	ECF_JOB_CMD ..	12, 16, 23, 28, 29, 30, 32, 59, 80, 82
Altering the clock .....	56	ECF_JOBOUT.....	34
Automatic rerun .....	35	ECF_KILL_CMD.....	29
CDP .....	79	ECF_LISTS .....	28
Check point files.....	74	ECF_LOG.....	32
Checking ecFlow Port Numbers .....	71	ECF_MICRO .....	23, 31
Checking ecFlow running.....	73	ECF_OUT .....	31
Checking Job Generation .....	68	ECF_PORT.....	28, 29
Checking the suite definition .....	67	ECF_SCRIPT.....	34
CLI Scripting in batch.....	79	ECF_TRIES.....	30, 35
Comment.....	16, 22	ECF_TRYNO .....	35
Control Structures.....	66	ecFlow Availability.....	81
Creating a job file.....	24	ecflow_client commands .....	7, 21
Debugging		ecflowview .....	84
Definition files .....	14	Collector .....	93
ECFLOW scripts .....	14	ecfmicro .....	17
Definition files .....	9	Error Handling .....	70
Dependencies .....	46	Flags used by ecFlow.....	82
Using together.....	53	Handling Dummy Tasks.....	69
ECF_CHECK .....	32	History of ecFlow .....	4
ECF_FETCH.....	31	Include files .....	16, 20
ECF_FILES.....	17, 31	Log files .....	73
ECF_HOME.....	28, 30	Log server.....	76
ECF_INCLUDE .....	31	manual .....	17
ECF_JOB.....	34	Manual pages .....	18

<b>Moving suites .....</b>	<b>14</b>	edit .....	59
<b>Pre-processor.....</b>	<b>16</b>	endfamily.....	46
<b>Python API.....</b>	<b>62</b>	endsuite.....	46
<b>Python Suite definition .....</b>	<b>62</b>	endtask .....	46
<b>Script writing guidelines .....</b>	<b>10</b>	event.....	39, 63
<b>Simulation of a running suite.....</b>	<b>69</b>	extern .....	60
<b>smsping .....</b>	<b>73</b>	family .....	38, 62
<b>Start up files .....</b>	<b>73</b>	label .....	42, 63
<b>Starting the ecFlow Server.....</b>	<b>71</b>	late.....	60, 66
<b>Start-up scripts .....</b>	<b>80</b>	limit.....	44, 63
<b>Status</b>		meter .....	41, 63
Family .....	8	repeat .....	61, 65
Task .....	7	show .....	79
<b>sterr and stdout .....</b>	<b>76</b>	suite .....	38, 62
<b>Stopping ecFlow server .....</b>	<b>72</b>	task .....	39, 62
<b>Suite</b>		time .....	51, 64
Definition.....	38	today.....	52, 64
Definition files .....	9	trigger .....	47, 63
<b>Suite commands</b>		<b>Tasks</b>	
Adding variables.....	63	Running remotely.....	12
autocancel.....	55, 65	<b>Terminology .....</b>	<b>5</b>
clock .....	56, 65	<b>Time Critical tasks.....</b>	<b>11</b>
complete .....	57, 63	<b>Using ecFlow .....</b>	<b>9</b>
cron .....	52, 64	<b>Variables .....</b>	<b>34</b>
date .....	49	ECFLOW .....	26, 29
day.....	50	ECFLOW client .....	28
defstatus.....	59, 65	ECFLOW environment .....	27

Generated .....31  
Inheritance .....26

**Writing scripts ..... 10**  
**Zombies ..... 3, 96**