# EcFlow

# Background

For almost three decades ECMWF has used the SMS (Supervisor Monitoring Scheduler) package to control the workflow for the Centre's operational models and systems. Written at ECMWF, it allows jobs design, submission and monitoring both in the research and the operations departments, providing common tools for scientists, analysts and operators, to cooperate.

# Replacement

The replacement, ecFlow, will supersede SMS in the near future. It is a complete rewrite from scratch using object oriented methodology and modern standardised components and will act as a comprehensive replacement for SMS. The rewrite will help improve maintainability, allow easier modification and introduce object orientated features.

Like its predecessor, ecFlow is a work flow package manager that enables users to run a large number of programs (with dependencies on each other and on time) in a controlled environment. It provides tolerance for hardware and software failures, combined with good restart capabilities.

EcFlow runs as a server receiving requests from clients. The command line interface, the graphical interface (ecFlowview), tasks scripts and the Python API (application interface) are the clients. The server is based on C++/boost ASIO and uses TCP/IP for communication. Multiple servers can be run on the same hardware. EcFlow submits tasks (jobs) and receives acknowledgements from tasks via child commands embedded in the scripts. EcFlow stores the relationship between tasks, and is able to submit tasks dependent on the status of other tasks and trigger events, such as time.

The command line interface for ecFlow allows the suite definition to be loaded and retrieved from the server. It provides a rich set of commands for communication with the server and provides similar capabilities to SMS. The Python API allows the entire suite definition structure to be specified and loaded into the server. A suite is a collection of interrelated tasks and is described by a definition file. The Python API also provides functionality for client to server communication. In addition it allows checking of the suite, testing the defined interrelations between tasks and other references and limits.

Another feature of ecFlow is the simulator. It can validate suites before the real job submission. The simulator can be run without the need for scripts or the server. For suites which are known to complete, it also helps to detect programming deadlocks.

EcFlow provides user manual, online tutorial, Python API and reference documentation.

# Other features of ecFlow

The initial graphical interface to ecFlow, ecFlowview, is used to visualise and manage the suite hierarchy. Currently it is based on and almost identical to the SMS graphical interface XCDP. The interface will be replaced in the near future, using more modern components.

EcFlow is written to be as platform independent as possible, by the use of standardised libraries. We have built and are testing ecFlow on the following operating systems: Linux (SUSE 10.3/11.3 and SLES 11), HP-UX (11.23) and AIX (5.3)

The software has been built with a rich suite of regression tests. These tests validate the expected behaviour and detect regressions in functionality and performance. They are run on each of the supported platforms. EcFlow has been written and developed in C++ and Python.

# Similarities to SMS

EcFlow has similar functionality and structure to SMS. Child commands are embedded in submitted scripts that communicate with the ecFlow server. It supports variable inheritance in the same way as SMS and also has white lists for authentication of read/write client request. It shares the ability to dynamically add or remove whole suites on the fly using the client level interface or the Python API.

EcFlow scripts are similar to those written for SMS. The default extension has been changed to reflect the software change with the “.ecf” extension used rather than “.sms”. EcFlow still has a pre-processor to handle includes files and variable substitution, though SMS variables have been replaced with ECF\_ labelled equivalents, e.g. The SMS variable SMSHOME is replaced with ECF\_HOME. These changes are easy to migrate with a script and thus preserve the investment made in the initial software development.

# Difference with SMS

With the use of standardised libraries and by designing it in an object orientated way, maintenance and enhancement of ecFlow is easier. EcFlow is built from the ground up in C++. It is fully test driven, with a large set of unit and regression tests built in. EcFlow also works on 64 bit operating systems, like AIX, without the need to compile in 32 bit mode.

Whilst SMS provided a custom scripting language, ecFlow provides integration with the Python language, and Python is used as a replacement for CDP/text interface. The python API provides two main functions. It allows the suite definition to be built and provides functionality for interaction between the client and server. This allows the suite definition to be loaded and retrieved from the server.

However, any language can be used to create the suite definition file as it has a published format. The format is similar to the SMS format, to provide some compatibility. The new format lacks conditionals, looping structures and function definitions, since that capability is provided by the Python extension.

When a task loses communication with its submitting server (SMS or ecFlow) it is called a zombie. EcFlow has been extended to allow for customisable handling of zombies. A new attribute specifies how zombies are handled by the client or server. For example, we can add a zombie attribute so that child label commands do not block the scripts. This can be added at different levels in the node tree hierarchy.

EcFlow has also been designed with better error checking, such as:

* Trigger expressions checking is improved. When a definition is loaded external references are checked in the server. This provides early warnings on trigger expression that never evaluate.
* Job generation can be checked before loading the definition into the server. This will check that script can be located, that the pre-processor can expand the include files and that variable substitution works. Recursive include files are also detected.
* Limit checking. References to limits are checked to ensure they exist. Also the tokens specified on the in-limit are checked to make sure that they are not larger than the limit size.
* Correct by construction approach is used when building the suite definition with the Python API.
* The abort child command, now allows you to provide a reason for the abort allowing for better handling of script problems.
* Definition files can be simulated without the need for underlying scripts or server.
* Scripts provide facilities which allow for a more automated way to kill processes.

# Incompatibilities

EcFlow is a replacement for SMS providing similar capabilities. Hence not all the functionality will be present in the first release. Currently auto-restore, auto-migrate and aliases have not been implemented. This will be added in a future version. SMS constructs owner, action, text, abort are no longer supported.

The replacement does not provide, looping and conditional statements or function definitions. However, since we will have a defined file format, any language can be used to generate the structure of the definition file. Python integration replaces CDP and will provide looping and conditional statements and functions.

# Migration

EcFlow will first be validated by the operations department at ECMWF. Once this has been done and a level of stability established, it will be opened up to other departments internally and then eventually the member states.

The task scripts and header files can be re-used with a little modification, such as renaming of SMS variables to ecFlow variables and of the child commands.

There are several routes available for migrating the definition file. The simplest route would involve dumping a flat file from SMS. This is directly readable from ecFlow since there are no conditionals or loops. A completely automated migration from CDP based definition file to ecFlow based Python scripts will not be possible. It should be noted that Python is fully object oriented language, and in order to take full advantage, it's recommended that the CDP scripts are completely re-written. This will allow for more compact and maintainable representation.

# Correct by Construction

The following examples show usage of the new Python API. The API supports a correct by construction approach. For example adding tasks of the same name at the same level, will throw a RuntimeError exception:

Example 1

import ecflow  
 defs = ecflow.Defs()   
suite = defs.add\_suite("s1")   
suite.add\_task("t1")   
suite.add\_task("t1") # RuntimeError exception thrown

>> RuntimeError: Add Task failed: A task of name 't1' already exist on node SUITE:/s1

Example 2

Adding dependencies like dates are also checked:

import ecflow   
defs = ecflow.Defs()   
suite = defs.add\_suite("s1")   
task t1 = suite.add\_task("t1")   
t1.add\_date(1,14,2007) # day,month,year, month is not valid

>> IndexError: Invalid Date(day,month,year): the month >=0 and month <= 12, where 0 means wild card

# Checking

## Expression Checking:

Some checking has to be deferred until the definition is fully defined. Here is a simple example, showing the checking of trigger expressions:

import ecflow   
defs = ecflow.Defs()   
suite = defs.add\_suite("s1");   
suite.add\_task("t2")   
suite.add\_task("t1").add\_trigger("t2 == active)")   
 assert len(defs.check()) != 0, "Expected Error: miss-matched brackets in expression."

## Job checking:

Job creation is the process of locating an '.ecf' script corresponding to a task and then generating a job file. This can also checked before a definition is loaded into the server using the Python API.

# Generate jobs for the \*ALL\* tasks in the definition given by variable 'defs'   
# and print errors to standard out.   
 import ecflow  
defs = ecflow.Defs()   
suite = defs.add\_suite("s1");   
suite.add\_task("t1")   
job\_ctrl = JobCreationCtrl()   
defs.check\_job\_creation( job\_ctrl )   
print job\_ctrl.get\_error\_msg()

Job control provides additional functionality to control which nodes are generated and control over the directory used for job generation.

## Dead Lock Checking:

Simulation allows a suite definition to be checked without the need for scripts or a server. By default the simulation will run for a year, before quitting. This can take a couple a seconds to a few minutes depending on the complexity of the suite definition. However, it is most useful where we have a definition which is known to complete (i.e. has no infinite repeat or cron attributes). Here is an example which will cause a deadlock that is detectable by the simulator.

import os   
 from ecflow import \*

defs = 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();   
assert len(theResult) != 0, "Expected simulation to return errors, but found none"   
print theResult

os.remove("defs.depth") # provides reason why simulation could not complete os.remove("defs.flat") # provides reason why simulation could not complete

Early checking of the suite definition will help to speed up the development of suites.

# Client server communication

Once the definition has been created (by text or Python API) it can be loaded into the server and played. This can be done in two ways.

1. On the command line:

> export ECF\_PORT=3141   
> ecflow\_client --restart   
> ecflow\_client --load=/path/to/defs/file.def   
> ecflow\_client --begin=s1

1. Directly from the Python API:

import ecflow

# create the defs, in memory   
defs = ecflow.Defs()   
suite = defs.add\_suite("s1")   
for i in range(1,7) :   
 fam = suite.add\_family("f" + str(i))   
 for t in ( "a", "b", "c", "d", "e" ) :   
 fam.add\_task(t);

# Example of loading an already defined definition   
# defs = ecflow.Defs("/path/to/defs/file.def")

# Don't load suite into server, until job creation works   
job\_ctrl = JobCreationCtrl()   
defs.check\_job\_creation( job\_ctrl )   
assert len(job\_ctrl.get\_error\_msg()) == 0, job\_ctrl.get\_error\_msg()

# Load definition into the server   
ci = ecflow.Client();   
ci.set\_host\_port("localhost","3141")   
 try:   
 ci.restart\_server() # if server halted restart it.   
 ci.load(defs) # load the definition into the server   
 ci.begin\_suite("s1") # play the definition   
except RuntimeError, e:   
 print "failed: " + str(e);

## A more complex example

Here is a more complex example taken from the online ecFlow tutorial. It is based on the Data Acquisition exercise from the current SMS online tutorial. It describes a data acquisition system, where every hour we receive data from Exeter, Toulouse and Offenbach. Every three hours we receive data from Washington. Once a day we receive data from Tokyo, every Monday from Melbourne and every first of the month from Montreal.

import os   
import ecflow

defs = ecflow.Defs()   
suite = defs.add\_suite("data\_aquisition")   
suite.add\_repeat( ecflow.RepeatDay(1) )   
suite.add\_variable("ECF\_HOME",os.getenv("HOME") + "/course") suite.add\_variable("ECF\_INCLUDE",os.getenv("HOME") + "/course") suite.add\_variable("ECF\_FILES",os.getenv("HOME") + "/course/data") suite.add\_variable("SLEEP","2")   
for city in ( "exeter", "toulouse", "offenbach", "washington", "tokyo", "melbourne", "montreal" ) :  
 fcity = suite.add\_family(city)   
 fcity.add\_task("archive")   
 for type in ( "observations", "fields", "images" ):   
 type\_fam = fcity.add\_family(type)   
 if city in ("exeter", "toulouse", "offenbach"): type\_fam.add\_time("00:00 23:00 01:00")   
 if city in ("washington") : type\_fam.add\_time("00:00 23:00 03:00")   
 if city in ("tokyo") : type\_fam.add\_time("12:00")   
 if city in ("melbourne") : type\_fam.add\_day( "monday" )   
 if city in ("montreal") : type\_fam.add\_date(1,0,0)

type\_fam.add\_task("get")   
 type\_fam.add\_task("process").add\_trigger("get eq complete”)  
 type\_fam.add\_task("store").add\_trigger("get eq complete")

# Load/Play the created defs into the server  
 ci = ecflow.Client();   
ci.set\_host\_port("localhost","3141")

try:   
 ci.restart\_server() # if server halted restart it.   
 ci.load(defs) # load the definition into the server   
 ci.begin\_suite("data\_aquisition") # play the definition   
except RuntimeError, e:   
 print "failed: " + str(e);

## Technology Used

Building ecFlow uses the latest C++ compilers. Currently we have tested building with the following compilers: gcc version 4.2.1/4.5 on Linux 32/64(SUSE 10.3/11.3), aCC-A.06.20 on HP-UX and xlC version 11.1 on AIX(ibm\_power6/rs6000). We have also made use of Python versions 2.1-2.7.

To support the generation of online documentation we use Sphinx-poco. This allows Python samples to be compiled separately and to be used in the documentation. It also allows the Python API to be documented via the doc strings.

We use the Boost C++ libraries. These are a set of free peer-reviewed portable C++ source libraries in preparation for inclusion in future C++ standard libraries. The Boost-ASIO library is used to provide the core of the client-server implementation. It also provides a deadline timer for polling and support for time outs. We also make use of Boost-Python library to enable seamless interoperability between C++ and the Python programming language. Boost-Program option is used to parse the client and server program options and provide the corresponding help strings. Boost-Spirit provides the parsing for the trigger and complete expressions. The abstract syntax tree is then created from the spirit nodes.

Boost-Test library is used for unit and regression tests of the C++ and Python code. The Boost-Date-Time library is used in the suite calendar, time based attributes and polling and the Boost-File System library is used for file query, in a platform independent manner.

To build the project we use the bjam command-line tool that drives the Boost Build system. It provides the build mechanism that allows ecFlow to be built in the same way across different platforms. It also allows debug, release and profile builds without knowledge of the compiler options on each platform