

**Earth System Science Workbench**

---

# **Lab Notebook Client Perl API and LN Console**

---

User's Manual 1.0ms

April 2000 revision

**University of California, Santa Barbara**

**ECOlogic Corporation**

<b>1</b>	<b>INTRODUCTION</b>	<b>4</b>
<b>2</b>	<b>NOMENCLATURE</b>	<b>4</b>
2.1	Predefined Templates	5
<b>3</b>	<b>ESSW CONFIGURATION INFORMATION</b>	<b>7</b>
3.1	How ESSW finds the configuration file and instance	7
3.1.1	<i>How XMLBLD Daemon and ESSW Console find the configuration file</i>	7
3.1.2	<i>How XMLBLD Daemon and ESSW Console find the instance within the configuration file</i>	7
3.1.3	<i>How Perl API programs find the Config file and the Instance name</i>	8
3.2	Format of the configuration file	8
3.3	Configuration File Reference	9
3.3.1	<i>Database-connection related configuration: needed by the console and xmlbld.</i>	9
3.3.2	<i>Error Logging configuration information</i>	10
3.3.3	<i>XMLBLD daemon configuration information</i>	10
3.3.4	<i>Other Optional configuration information.</i>	11
<b>4</b>	<b>THE ESSW CONSOLE</b>	<b>12</b>
4.1	Starting the console	12
4.2	Console command reference	12
4.2.1	<i>Quick Reference</i>	12
4.2.2	<i>Detailed console reference</i>	13
4.2.2.1	add	13
4.2.2.2	alter	14
4.2.2.3	describe	14
4.2.2.4	history	15
4.2.2.5	list	15

4.2.2.6	newversion	15
4.2.2.7	quit	16
4.2.2.8	remove	16
4.2.2.9	select	16
4.2.2.10	setverbose	17
4.2.2.11	sh	17
<b>5</b>	<b>THE XMLBLD DAEMON</b>	<b>18</b>
5.1	Starting the XMLBLD daemon	18
5.2	XMLBLD Daemon Utility programs	18
5.2.1	<i>syscheck.pl</i>	18
5.2.2	<i>shutdown.pl</i>	19
5.2.3	<i>loggingon.pl</i> and <i>loggingoff.pl</i>	19
5.3	Troubleshooting the XMLBLD daemon	19
<b>6</b>	<b>THE PERL API</b>	<b>20</b>
6.1	ESSW Perl program structure	21
6.2	Perl API Reference	22
6.2.1	<i>addValue</i>	22
6.2.2	<i>attachNote</i>	22
6.2.3	<i>beginXMLBld</i>	23
6.2.4	<i>checkMetadataId</i>	24
6.2.5	<i>checkXML</i>	24
6.2.6	<i>createMetadata</i>	25
6.2.7	<i>createExperimentMetadata</i>	25
6.2.8	<i>createExpStepMetadata</i>	26
6.2.9	<i>closeLogFile</i>	26
6.2.10	<i>closeMetadata</i>	27
6.2.11	<i>endXMLBld</i>	27

6.2.12	<i>getMetadataAsString</i>	27
6.2.13	<i>esswlink</i>	28
6.2.14	<i>getFieldNames</i>	28
6.2.15	<i>logToFile</i>	29
6.2.16	<i>metadataNameSearch</i>	29
6.2.17	<i>query</i>	30
6.2.18	<i>query_hash</i>	31
6.2.19	<i>registerExpStepInputs, registerExpStepOutputs</i>	32
6.2.20	<i>registerExperimentInputs, registerExperimentOutputs</i>	32
6.2.21	<i>saveToFile</i>	33
6.2.22	<i>saveToDB</i>	34
6.2.23	<i>setErrorHandlingMode</i>	34
6.2.24	<i>shutdownServer</i>	35
6.2.25	<i>sysCheck</i>	35
6.2.26	<i>setVerbose</i>	36
6.3	<i>Prototype Lineage Perl API and Commands</i>	36
6.3.1	<i>Prototype Lineage Perl API</i>	37
6.3.1.1	<i>getStepForData</i>	37
6.3.1.2	<i>getInputsForStep</i>	37
6.3.1.3	<i>getInputsOfDepthForData</i>	38
6.3.1.4	<i>getOutputsOfDepthForData</i>	39
6.3.1.5	<i>getStepsForwardOfDepthForData</i>	39
6.3.1.6	<i>getExperimentForData</i>	40
6.3.1.7	<i>getExperimentForStep</i>	40
6.3.1.8	<i>getExpObjects</i>	41
6.3.1.9	<i>getLineage</i>	42
6.3.3	<i>Perl Lineage Scripts</i>	44

6.3.3.1	getExperimentForData.pl x	44
6.3.3.2	getExperimentForStep.pl x	44
6.3.3.3	getInputsOfDepthForData.pl x y	44
6.3.3.4	getOutputsOfDepthForData.pl x y	44
6.3.3.5	getStepsForwardOfDepthForData.pl x y	45
6.3.3.6	cmp.pl x y	45
6.3.3.7	getLineage.pl argv0 argv1 argv2 argv3 argv4	45
6.4	Metadata Standards	46
6.4.1	getDTDForVocabTerm	46
6.4.2	getTagsForKeyword	47
6.4.3	getTagsForDTD	48
<b>7</b>	<b>CHANGES SINCE LAST RELEASE</b>	<b>49</b>

# 1 Introduction

The Console and Perl API are important components of the Earth System Science Workbench. This document describes how to use the console and the Perl API, and provides reference information.

The Perl API is a suite of Perl functions for constructing science-metadata and submitting it to the ESSW database. The Perl API also provides methods for extracting metadata from the database. The Perl API constructs metadata in the format language XML. This document does not describe XML. Further information on XML can be found at <http://www.xml.com>

The ESSW Console is a simple command-line program for monitoring the ESSW database, querying the database, telling the database about new kinds of metadata, and submitting new science metadata (and other information) to the database.

There are other parts of the ESSW Project. These are documented elsewhere.

## 2 Nomenclature

ESSW database is designed to store information about science data, and also information about the computer programs that were run to create that data. We use some common science terms in somewhat specialized ways, so here is a brief description of some of those terms, and how we use them in ESSW:

Word	Our Meaning
<b>Model</b>	A complete set of one or more science computer programs, which read and generate science data.
<b>Experiment</b>	An instance of a model: an actual run of a <b>set</b> of programs in that model.
<b>experiment-step or expstep</b>	A single actual run of a computer program in an experiment. In ESSW, we can store metadata about both data and about the experiment steps that created that metadata.
<b>Metadata</b>	In ESSW, metadata means an XML document that stores useful attributes about one or more pieces of science data.
<b>Data</b>	This one is pretty poorly defined: basically, something referred to by a metadata, but is not an expstep. In practice, data means the files that are input or output of an expstep or an experiment.
<b>Dtdlib</b>	An XML DTD: A set of interrelated XML tags, with no particular ordering, submitted to ESSW. A dtdlib has no <i>root tag</i> .
<b>Template</b>	A specific subset of XML tags in a dtdlib that you want to use for a specific kind of metadata. Every metadata submitted to ESSW must conform to a previously submitted template. A template is a dtdlib with a <i>root tag</i> .
<b>Note</b>	A text annotation about some metadata. It is similar to notes taken in a scientist's lab notebook.
<b>Science Object</b>	This is a very wide definition that applies to any type of metadata stored into ESSW. Models, notes, experiments, experiment steps, input data, output data - are all science object. This term originates from the implementation - where all instances of all types in ESSW reside in tables that are derived from a tables called Science Object.

The ESSW Console and Perl API allow the storage of XML-based metadata. In order to do this, you need to submit dtdlibs, templates, and metadatas to the ESSW. The following table describes these in more detail:

Name	Purpose	How it uses XML
<b>dtdlib</b>	Submitting a dtdlib to ESSW defines and tells the ESSW database about a set of metadata names (tags) that you wish to use in your science database. Anyone using ESSW (including you, of course) can re-use those names in their <i>templates</i> . Including an existing dtdlib in a new dtdlib allows the new dtd lib to use metadata names (tags) from the initial dtdlib without redefining them.	A dtdlib is an XML <i>Document Type Declaration</i> , or DTD. In XML, the metadata names you define are called <i>tags</i> .
<b>template</b>	Submitting a template tells ESSW about a specific subset of tags in a dtdlib that you want to use for a specific kind of metadata. There are three special predefined templates: model, experiment, and note. In addition, the ESSW console lets you submit your own templates. A template is obtained from a dtdlib by selecting a root element (tag).	A template is defined by two things: (1) a dtdlib, and (2) a single tag in that dtdlib chosen to be the <i>root tag</i> . The name of the template is the name of its root tag.
<b>metadata</b>	A metadata is a chunk of useful information that conforms to the pattern of a previously submitted template. Most often, a metadata is scientific metadata about some science data. However, in ESSW, notes, models, experiments, and expsteps are also represented as metadatas.	In ESSW, a metadata is an XML document. Its <i>root-tag</i> must be an existing <i>template</i> in the ESSW database. A metadata consists of the XML document, plus an optional list of data to which the metadata refers.

The ESSW Console program allows different versions of the same dtdlib, and also different versions of the same template. See the *newversion* command in the ESSW Console Reference section below. Metadatas do not have versions in ESSW.

## 2.1 Predefined Templates

The ESSW has several special predefined templates. They are found in \$ESSWHOME/bin/. They are:

Template Name	Purpose	Example tags in that Template	How to access/create Metadata for this Template
<b>model</b>	Store general information about science models. See the definition of model in the Nomenclature section.	brief, url, DAG_url	Generate a model xml file, and submit it using the ESSW console.
<b>experiment</b>	Stores the existence of a new experiment, the date it occurred, and what model it ran under.	model_id	Use the Perl API function createExperiment.
<b>Note</b>	Stores a general note about a piece of science metadata. Meant to simulate the information typically written in a scientist's lab notebook.	subject, body, crossed_out, importance, refers_to_obj	Use the notebook Web interface, or use the addNote Perl API function.

## 3 ESSW Configuration Information

This section describes the configuration of the ESSW database application components, and how you can configure them to meet your needs.

The ESSW consists of three basic applications: a command line console program, a perl module for generating metadata, and a web-based notebook application. All of these programs read from the `essw` configuration file.

The ESSW configuration has been designed to make it easy for new users to attach and run. Most users should only have to set one environment variable: `$ESSWHOME`, and add `$ESSWHOME/bin` and the `java1.2` executable to their paths. This should be enough to get the default production environment for ESSW. ESSW Administrators will have to read this section carefully to see how to set up all the needed configuration information.

The configuration file is divided into sections. Each section contains `key=value` pairs that define a single instance of ESSW. This way, you can have multiple instances of the ESSW running, but you only have to manage a single configuration file. When you start the XMLBLD daemon, and the ESSW Console, you need to specify both the location of the configuration file, and the instance name in that configuration file.

### 3.1 How ESSW finds the configuration file and instance

#### 3.1.1 How XMLBLD Daemon and ESSW Console find the configuration file

The ESSW Console and XMLBLD daemon try three ways to find the configuration file. First, the applications look for the `-e <filename>` command line option. If found, that filename is used as the configuration file path. If not found, the applications look for a Unix environment variable called `$ESSW_CONFIG_FILE` which represents the full path to the configuration file. If neither the `-e` option nor the environment variable is found, the daemons look for the default configuration file, called:

`$ESSWHOME/essw.config`

At this point, if the `$ESSWHOME` environment variable is not set, or if `$ESSWHOME/essw.config` is not found, then the applications will abort with an error message.

We recommend putting your configuration file in the default location for most users. Ideally, only ESSW administrators should have to use the `-e` option or the `ESSW_CONFIG_FILE` environment variable.

The applications are aborted if the configuration file is not found, or if errors are found in the file.

#### 3.1.2 How XMLBLD Daemon and ESSW Console find the instance within the configuration file

The ESSW Console and XMLBLD daemon try three ways to find the instance (section) within the configuration file. First, the applications look for the `-s <instancename>` command line option. If found, that instance name is used. Second, the applications look for a Unix environment variable called `$CONFIG_FILE_SECTION` which represents the exact name of the instance. If neither the `-s` option nor the environment variable is found, the daemons look for the default instance name, called:

PRODUCTION

If that instance name is not found in the configuration file, then the applications abort with an error message.

We recommend using two instances: PRODUCTION and TEST. Your PRODUCTION instance will be your permanent persistent storage for data and metadata and lineage tracking. The TEST instance will be another database that is just for scientists to test their Perl API scripts.

### 3.1.3 How Perl API programs find the Config file and the Instance name

Users of the Perl API must call the Perl API function `beginXmlBld(...)` in order to read the configuration file and connect with the database. The `beginXmlBld()` function takes optional arguments that allow the user to specify an instance-name and a configuration-file-path. If the configuration file path is not given, the default is `$ESSWHOME/essw.config`. If the instance-name is not given, the default is PRODUCTION. Please see the PerlAPI reference documentation for a complete description of all the arguments to the `beginXmlBld()` function.

## 3.2 Format of the configuration file

Here is an example of a configuration file that defines two ESSW instances: sections labeled PRODUCTION, and TEST:

```
[PRODUCTION]
  PASSWORD=mysecret
  USERNAME=dbuser1
  DATABASEHOST=foo.bar.baz.edu
  DATABASEPORT=5555
  DATABASESERVER=my_informix_servername
  DATABASENAME=essw_production_db
  OPENDBCONNECTIONS=5
  MAXDBCONECTIONS=10
  MESSAGELOG="/tmp/esswlog"
  CONSOLELOG=consolelog
  # Note that the quotes can be used as well:
  XMLBLDLOG="production_xmlbldlog"
  UNIT="production"
  DEBUGSTATUS="NoDebug"
  DAEMONPORTNUMBER=8233
  DAEMONHOST=foo.bar.baz.edu
[TEST]
  PASSWORD=mysecret
  USERNAME=dbuser1
  DATABASEHOST=foo.bar.baz.edu
  DATABASEPORT=5555
  DATABASESERVER=my_informix_server
  DATABASENAME=essw_test_database
  # This is how you do a comment!
  OPENDBCONNECTIONS=2
  MAXDBCONECTIONS=4
  MESSAGELOG="/tmp/esswlog"
  CONSOLELOG=test_consolelog
  XMLBLDLOG="test_xmlbldlog"
  UNIT="essw test"
  DEBUGSTATUS="MaxDebug"
  DAEMONPORTNUMBER=8244
  DAEMONHOST=foo.bar.baz.edu
```

You can use any instance names you like. You can have as many instance sections as you like. The instance names and keywords are case-sensitive, so be sure to reference them properly when using them. Right now you cannot attach more than one section name to the same set of key=value pairs. The default instance name is PRODUCTION, so be sure you have a section with that name somewhere in your configuration file. The default configuration file is \$ESSWHOME/essw.config, so for simplicity, you should always have a working configuration file in that location.

Comment lines in the configuration file start with the # character and go to the end of the line. Comments must be on their own line. You should surround the values with quoted whenever special characters need to appear in the value. The special characters are: "/", "\", ", ";", ":".

Note for Perl API users: the Perl API needs *only two* key=value keywords from the configuration file! Users of only the Perl Api could use a minimal configuration file. In particular, they do not need the internal database and password information. The information in the configuration file indicates which Java daemon the client wants to connect to.

Here is an example of the minimal config file that an outside Perl API user could use:

```
[PRODUCTION]
  DAEMONPORTNUMBER=8233
  DAEMONHOST=foo.bar.baz.edu
[TEST]
  DAEMONPORTNUMBER=8244
  DAEMONHOST=foo.bar.baz.edu
```

### 3.3 Configuration File Reference

The configuration keywords support a variety of purposes, so we document them here in several sections

#### 3.3.1 Database-connection related configuration: needed by the console and xmlbld.

<i>Keyword</i>	<i>Purpose/Meaning</i>
PASSWORD	Password to use to connect to the database.
USERNAME	Username to use when connecting to the database. This does not have to be the userid under which the console or xmlbld is running.
DATABASEHOST	hostname of the machine where the database server is running.
DATABASEPORT	standard port on which the database is listening.
DATABASESERVER	Name of the Informix server instance to attach to.
DATABASENAME	Name of the database to attach to
OPENDBCONNECTIONS	Number of continuously open database connections to maintain between xmlbld daemon and the database server. (Note: Regardless of this value, the console only opens one connection.)
MAXDBCONNECTIONS	Maximum number of database connections to maintain between xmlbld daemon and the database. Be sure that MAXDBCONNECTIONS >=

OPENDBCONNECTIONS. The extra connections are opened on an 'as needed' basis, and are closed right away.
---

### 3.3.2 Error Logging configuration information

This information is needed only by the console and the xmlbld daemon.

The paths for the files where state and debug information is written by ESSW modules can be full paths or relative paths to the directory in which the programs are started.

<b><i>Keyword</i></b>	<b><i>Purpose/Meaning</i></b>
MESSAGELOG	Logfile location for the Annotation tool (Note: if CONSOLELOG and/or XMLBLDLOG are not set, this will be used as the log location for the console and xmlbld, too.)
CONSOLELOG	Logfile location for the console log
XMLBLDLOG	Logfile location for the xmlbld daemon log
UNIT	A simple name to put at the beginning of each log message. Usually this is just the name of the application. For example: xmlbld-test This is helpful in figuring out what program produced a particular log file.
DEBUGSTATUS	Level of logging to produce. There are currently two possible values to use here: "NoDebug" or "MaxDebug". The NoDebug does not produce any debug information, the MaxDebug logs verbose debug information.

Note: If problems are encountered before the logfiles can be opened, then error information will be logged in the following location:

`/var/tmp/ESSW_ERROR.LOG.<youruserid>`

### 3.3.3 XMLBLD daemon configuration information

Only the xmlbld daemon and the Perl API need this information.

<b><i>Keyword</i></b>	<b><i>Purpose/Meaning</i></b>
DAEMONHOST	Hostname where the daemon is running. Note: this keyword is used only by the Perl API, not by the daemon itself. The daemon is started on the desired machine.
DAEMONPORTNUMBER	Port on which the xmlbld daemon is listening for Perl API requests.

### 3.3.4 Other Optional configuration information.

These are documented here for completeness. In general, you should not need to use them:

<b><i>Keyword</i></b>	<b><i>Purpose/Meaning</i></b>
LOB_MEMORY_CACHE	Default is 65535. XMLs and DTDs larger than this size will be written into temporary files when read from the database. This will be visible to the user as annoying files left sitting around their directories.
DATABASEDRIVER	internal JDBC driver to use to connect. Default is com.informix.jdbc.IfxDriver
DATABASEURL	internal JDBC string to use to connect with the database. This is just a shorthand for Java programmers, and overrides the following keywords: PROTOCOL, DATABASEHOST, DATABASEPORT, DATABASESERVER
PROTOCOL	Informix connection protocol. Default is "informix-sqli"

## 4 The ESSW Console

### 4.1 Starting the console

The console executable is found in `$ESSWHOME/bin/essw`.

To run the console, you will need to

1. have `$ESSWHOME` set properly to the ESSW home directory.
2. have `java1.2/bin` in your path, since the console is a `java1.2` program.
3. have `$ESSWHOME/bin` in your path.
4. have your config file setup properly. See the section this manual on ESSW configuration.

In the default situation, you shouldn't have to worry about finding the config file, if you have done those four steps.

Here's an example set of `cs`h commands to do this:

```
mars> setenv ESSWHOME /home/essw
mars> setenv PATH $ESSWHOME/bin:/usr/java1.2/bin:$PATH
mars> essw
ESSW 0.9.3 configuring from /home/essw/essw.config PRODUCTION instance.
Connected to database essw.
essw>
```

Here are the available command line options for starting the console:

`[-a filename] [-e configfile] [-s instancename]`

Command line option	Meaning
<code>-a &lt;filename&gt;</code>	have the ESSW console <code>&lt;a&gt;</code> utomatically read from a file rather than from <code>stdin</code> .
<code>-e &lt;filename&gt;</code>	to set the configuration file to read its <code>&lt;e&gt;</code> nvironment from. See section above on ESSW configuration to find other ways to set the location of the configuration file. If this option is not used, the default configuration file location is <code>\$ESSWHOME/essw.config</code>
<code>-s &lt;instancename&gt;</code>	set the SECTION of the configuration file to read from. If not used, the default instancename is <code>PRODUCTION</code> . A <code>PRODUCTION</code> section should be present in the configuration file.

### 4.2 Console command reference

The information here is taken from the console online help facility, which you can use by typing 'help' once the console is up and running.

#### 4.2.1 Quick Reference

Command	Purpose
Add	Add a new metadata, metadata template, or dtd library

	item to ESSW
Alter	Send alter table constraint command to the ESSW.
Describe	Get details about data objects, metadata templates, or dtdlibs.
History	List the commands that have been run
List	List names of available data objects, metadata templates, or dtdlibs
Newversion	Create a new version of an existing dtdlib item or metadata template. The newversion command is identical to the add command, except that it allows entities to have the same names as entities already in the database.
Quit	Quit the ESSW console program.
Remove	Remove a data object, metadata template, or dtdlib from ESSW
Select	Send arbitrary SELECT SQL query to the ESSW.
setverbose	Turn console logging on or off
sh	Execute a shell command without having to leave ESSW.
! <str&gt;< td=""> <td>Rerun the most recent command that starts with &lt;str&gt;</td> </str&gt;<>	Rerun the most recent command that starts with <str>
! or !!	Rerun the previous command.

## 4.2.2 Detailed console reference

### 4.2.2.1 add

#### Add new metadata, metadata template, or dtdlib to ESSW

To:	Run this console command:
add an dtd library item	add dtdlib <filename> [<dtdlibname>]
add a metadata template	add template <base-dtdlib> [-v <dtdlib-version>] [-r <root-tag-name>]
add xml metadata	add metadata <xmlfilename> [-n <metadata-name>] [-i datafile   -o datafile]*

#### How to use 'add dtdlib':

add dtdlib expects a complete valid XML External DTD file as input. If no dtdlibname is given, the filename (minus any .dtd suffix) will be used as the dtdlibname. To add a new version of an existing dtdlibname, use the 'newversion dtdlib' command.

#### How to use 'add template':

add template expects you to choose one tag in a DTD library to be the 'root' tag. This tag will form the root of a tree of child tags, much like a directory tree in a file system. If no root-tag-name is specified, ESSW will search the base-dtdlib for a tag with an attribute named 'root' or 'ROOT'. The highest level tag should be used as ROOT TAG. To avoid confusion, the highest level tag should also be the dtd file name. For example, for a file named model.dtd, the highest level tag should be called model. The highest level tag is the tag composed of all the other tags in the file.

If no dtdlib-version is specified, ESSW will use the latest version of base-dtdlib. If base-dtdlib is not found, ESSW will search for a file named <base-dtdlib> and will insert that into the dtd library repository, then use it as a basis for the new template.

### How to use 'add metadata':

add metadata is used to add new metadata and data to the ESSW. 'xmlfilename' is the name of an XML metadata file that you wish to add. 'metadata-name' is the name to assign to the metadata. If 'metadata-name' is not specified, then the filename will be used as the name, with a possible number attached to the end of it to guarantee uniqueness. Names of metadata must be unique within a given template. Note that the templatename is embedded in the metadata file, in its DOCTYPE declaration. Any number of actual science data files can be associated with the metadata. For each one, you should specify whether ESSW should store it IN the database (-i) or OUT of the database (-o).

NOTE: the -i, and -o options are not implemented yet.

### 4.2.2.2 alter

#### Send alter table constraint command to the ESSW

Usage: alter table <tablename> <rest of the alter SQL>

You can only alter tables that have your userid and are children of the SciObj table. This command is used by ESSW administrators to add foreign key references between child tables. The console uses this command internally when the predefined templates are created.

### 4.2.2.3 describe

#### Get details about metadatas, templates, or dtdlibs

To	Use this command:
describe dtdlibs:	describe dtdlib [where <where-clause>   [<name> [<version>]]
describe metadata templates:	describe template [where <where-clause>   [<name> [<typename> [<typeversion>]]]
describe objs:	describe metadata [where <where-clause>]   [<name> [[<template>] [<typeversion>]]]

#### How to describe dtdlibs:

With only a <name> given, this describes the latest version of the dtdlib with that name. With a <name> and <version> given, this describes the single dtdlib that matches that (name, version). With a valid SQL <where-clause>, this describes information about every dtdlib that meets the restrictions in the where clause.

#### How to describe metadata templates:

With only a <name> given, this describes the latest version of the type with that name.

With a <name> and <version> given, this describes the single template that matches that (name, version).

With a valid SQL <where-clause>, this describes information about every template that meets the restrictions in the where clause.

#### How to describe metadata:

With only a <name> given, this describes the object with that name.

With a <name> and <template> given, this describes the object that matches that (name, template).

With a valid SQL <where-clause>, this describes information about every metadata that meets the restrictions in the where clause.

#### 4.2.2.4 history

Lists the commands that have been run in this console session only.

#### 4.2.2.5 list

List names of available metadatas, templates, or dtdlibs

To:	Use this form of the command:
list dtdlibs:	list dtdlib [where-clause...]
list metadata templates:	list template [where-clause...]
list any objects:	list metadata [where-clause...]

##### How to list dtdlibs:

With no where-clause, this lists names and versions of every element-set in the library. With a valid SQL where-clause, this lists information about every element-set that meets the restrictions in the where clause.

##### How to list templates:

With no where-clause, this lists names and versions of every metadata-template in the library. With a valid SQL where-clause, this lists information about every metadata-template that meets the restrictions in the where clause.

##### How to list metadatas:

With no where-clause, this lists names, templates, and versions of every metadata object in database. With a valid SQL where-clause, this lists information about every metadata-template that meets the restrictions in the where clause.

#### 4.2.2.6 newversion

Create a new version of an existing dtdlib or template

To:	Use this version of the command:
create a new version of an dtdlib:	newversion dtdlib <filename> [<dtdlibname>]
create a new version of a template:	newversion template <base-dtdlib> [-v <dtdlib-version>] [-r <root-tag-name>]

The 'newversion dtdlib' and 'newversion template' commands are equivalent in every way to the 'add' commands, except that newversion will create new versions of these, rather than returning an error.

With the newversion command, if you submit a dtdlib or template whose name already exists in ESSW, that dtdlib or template will get created, but will be given a new version number.

metadatas don't have versions, so there is no 'newversion metadata' command.

### 4.2.2.7 quit

Quit the ESSW console program.

### 4.2.2.8 remove

Remove a metadata, template, or dtplib from ESSW

To:	Use this form of the command:
remove dtplib members:	remove dtplib <name> [<version>]
remove templates:	remove template <name> [<version>]
remove metadatas:	remove metadata -t <template> [-v <template-version>] [-n <name>]

#### How to remove dtlibs:

With a <name> given, this removes ALL the versions of the dtplib with that name. (Be Careful!) With a <name> and <version> given, this removes the single dtplib that matches that (name, version). NOTE: A dtplib can only be removed if it is

1. NOT referenced, named, or used in any other dtplib.
2. NOT referenced, named, or used in any metadata template.

#### How to remove templates:

With a <name> given, this removes ALL the versions of the template with that name. (Be Careful!) With a <name> and <version> given, this removes the single template that matches that (name, version). NOTE: An template can only be removed if it is

1. NOT referenced, named, or used by any metadata.

#### How to remove metadatas:

You can optionally specify a <template-version> or <name> of the template to further narrow what gets deleted. If neither <name> nor <template-version> is given, then this deletes every metadata of that template. NOTE: A metadata can only be removed if it is:

1. NOT referenced by any Link.
2. NOT referenced by any database foreign key reference.

### 4.2.2.9 select

Send arbitrary SQL query to the ESSW

Usage: select <arbitrary string: rest of the query>

This is just a simple way to send select statements to the database directly. No guarantees that the output will be nicely formatted. Only select statements are available (no update, insert, delete).

### 4.2.2.10 setverbose

Turn console logging on or off

Usage: setverbose off | <logfilename>

To turn off console logging, use 'setverbose off'. To change the logfile, or to turn logging on, use: setverbose <logfilename>. Note: this will wipe out any existing file by that name, even if it is the current log file. Note: This command affects this console program ONLY.

### **4.2.2.11sh**

**Execute a shell command without having to leave ESSW**

Usage: sh <shell-command and arguments...>

This is just a time-saver. Some shell commands don't work right. Unix only.

## 5 The XMLBLD Daemon

### 5.1 Starting the XMLBLD daemon

The xmlbld daemon executable is found in \$ESSWHOME/bin/xmlbld. In general, only essw administrators should have to start the xmlbld daemon.

To run the xmlbld daemon, you will need to

1. have \$ESSWHOME set properly to the ESSW home directory.
2. have java1.2/bin in your path, since the xmlbld daemon is a java1.2 program.
3. have \$ESSWHOME/bin in your path.
4. have your config file setup properly. See the section this manual on ESSW configuration.

In the default situation, you shouldn't have to worry about finding the config file, if you have done those four steps.

Here's an example set of csh commands to do this:

```
mars> setenv ESSWHOME /home/essw
mars> setenv PATH $ESSWHOME/bin:/usr/java1.2/bin:$PATH
mars> xmlbld &
ESSW 0.9.3 configuring from /home/essw/essw.config PRODUCTION instance.
mars>
```

Here are the available command line options for starting the console:

`[-c] [-f][-e configfile] [-s instancename]`

Command line option	Meaning
-c	try to <c>onnect to the database, but continue even if it fails
-e <filename>	to set the location of the config file. (-e for "environment") See section above on ESSW configuration to find other ways to set the location of the config file.
-f	<f>orce ESSW to run without the database
-s <instancename>	set the SECTION of the config file to read from. If not used, the default instancename is PRODUCTION.

### 5.2 XMLBLD Daemon Utility programs

The perl utility scripts are in \$ESSWHOME/bin.

#### 5.2.1 syscheck.pl

**Checks if the xmlbld daemon is running and display other useful information (ESSW version, daemon port number, log file name, log mode, user name, java version, database connection state, database host, daemon host).**

Usage: syscheck.pl [-s SECTION\_NAME],[-e CONFIG\_FILE]

[-e SECTION\_NAME] : This is optional. It specifies the section in the configuration file.

[*-i* CONFIG\_FILE] : This is optional. It specifies the location of the configuration file.

## 5.2.2 shutdown.pl

**Shuts down the xmlbld daemon.**

NOTE: You must have the DAEMONHOSTNAME configuration file keyword set to "localhost" for this to work. Remote shutdown has not been implemented yet.

Usage: shutdown.pl [*-s* SECTION\_NAME],[*-e* CONFIG\_FILE]

[*-s* SECTION\_NAME] : This is optional. It specifies the section in the configuration file.

[*-e* CONFIG\_FILE] : This is optional. It specifies the location of the configuration file.

Default section is PRODUCTION,

Default config file is \$ESSWHOME/essw.config.

## 5.2.3 loggingon.pl and loggingoff.pl

**Turn daemon logging on and off.**

Usage: Loggingon.pl [*-s* SECTION\_NAME],[*-e* CONFIG\_FILE]

[*-s* SECTION\_NAME] : This is optional. It specifies the section in the configuration file.

[*-e* CONFIG\_FILE] : This is optional. It specifies the location of the configuration file.

Usage: Loggingoff.pl [*-s* SECTION\_NAME],[*-e* CONFIG\_FILE]

[*-s* SECTION\_NAME] : This is optional. It specifies the section in the configuration file.

[*-e* CONFIG\_FILE] : This is optional. It specifies the location of the configuration file.

## 5.3 Troubleshooting the XMLBLD daemon

If xmlbld is unable to even find the configuration file, or does not have permissions to read that configuration file, or does not have permissions to open a log file, it will dump error messages in the file:

*/var/tmp/ESSW\_ERROR.LOG.<youruserid>*

Check that file for errors.

If xmlbld is at least able to read the config file, then check the XMLBLDLOG keyword in that config file to see where logging information is going. If that log file is empty, make sure the DEBUGSTATUS keyword is set to MaxDebug, and try it again. Make sure you have permissions to write to that log file.

You can also change the logging status by using the loggingon.pl and loggingoff.pl XMLBLD daemon utility program. You can use the syscheck.pl program to check up on your xmlbld daemon to see what port number it is listening on, and to make sure it has good connections to the database.

You should ALWAYS try to shut down the daemon using the shutdown.pl utility program. If this does not work, try shutting down with the kill command, then as a last resort, with the kill -9 command. Sometimes the kill command will leave the daemon-port-number in a hanging state. The only solution I know to this is to reboot the machine on which it's running. If any system hacker out there knows a better way, let us know!

## 6 The Perl API

The PERL XML Builder API is a set of PERL functions that should make it easier to generate Metadata in the XML format. The XML Builder will find your DTDs, and use them to check that you are building correct metadata.

We chose Perl because the idea is that much data processing would be done in an automated or semi-automated environment, and someone could write “Perl wrapper scripts” that would run before and after the data processing, to build the necessary metadata structures. Perl should be an easier language to use to read output text from a run, and produce needed metadata. For each science model, PERL cataloging tools will be developed, which will create and populate the appropriate metadata structures.

An implementation Note: XMLBld consists of two pieces: the Perl xmlbld module, and the xmlbld daemon. Here’s why: Perl is NOT a good language for either (1) communicating with databases, or (2) building complex structures like XML. So, the PERLAPI communicates with a XML-Builder daemon (written in Java) that does the actual Xml work, and communicates with the database. The Perl xmlbld.pm module contains functionality that allows client Perl scripts to connect to the xmlbld Java daemon and exchange information through a custom PERL socket protocol.

Right now there are just a few functions in the api. The main function you’ll use is ‘addValue’, which does the work of adding new metadata into the XML metadata structure.

## 6.1 ESSW Perl program structure

Here is a sort-of 'hello world' ESSW Perl program, so that you can see the basic structure:

```
#!/usr/local/bin/perl
# Perl compiler directive to add xmlbld.pm's location to the
# search path.
use lib $ENV{ESSWHOME} . "/perl";

# Perl compiler directive to use the xmlbld.pm API module.
use xmlbld;

## save messages in a log file, rather than to stdout.
logToFile("logging.txt");

# Connect to the daemon. Use the default instance ("PRODUCTION")
# and the default config file ($ESSWHOME/essw.config)
beginXMLBld ();

# Here's an example of a simple API call.
my @retval = sysCheck() or die "XML Errors:$xmlbld::errmsg";

# Print some results. See the sysCheck function in the User's Manual
# for more details.
print "I am running ESSW version $retval[0]\n";

# close log file.
closeLogFile();

## Disconnect.
endXMLBld();

exit;
```

Note the directive that was used above:

```
use lib $ENV{ESSWHOME} . "/perl";
```

which lets the program find the xmlbld.pm API module. ALL Perl programs must now include this line at the beginning. All new Perl modules should be placed in the \$ESSWHOME/perl directory.

In order for your perl-API programs to work, you will need:

- 1) set the \$ESSWHOME environment variable
- 2) have Perl available on your machine.
- 3) have the xmlbld daemon already running. Ask your ESSW administrator for help if a daemon is not already running. You can use the syscheck.pl utility program to check which daemon you are currently talking to.
- 4) have a properly setup configuration file, accessible by the Perl program. The configuration file is accessed by the beginXmlBld() call. See the Perl API reference below for details on how to tell beginXmlBld to find the configuration file and the right section in that configuration file.

Many example perl API programs can be found in the \$ESSWHOME/examples and in the \$ESSWHOME/test directories.

## 6.2 Perl API Reference

### 6.2.1 addValue

*addValue* (*metadata\_id*, *location*, *value*)

*metadata\_id*: string (from the call to createMetadata) id for the metadata the users wish to modify.

*location*: string showing where in the XML structure to save the value. This is actually a fully qualified XML tag.

*value*: The value to save at that location.

**Returns:** The string \*OK\* on success, or undef on errors. Also, on errors, an error message is placed in the variable \$xmlbld::errmsg.

**Possible Errors:** Using a non-existent xml tag. Using a non-existent metadata\_id. Attempting to insert the l+1'th element in an array without having first inserted the l'th. Attempting to insert into a tag declared EMPTY. Inserting a tag out of order.

**Explanation:** addValue is the main function for adding metadata to an XML metadata file. The location argument gives a path into the metadata -- a location where the value will be placed. When the metadata allows more than one tag with the same name, use square brackets [int] to specify which one you want to set. You must set these consecutively -- for example, XMLBld won't let you set the x.y[2] until you've set x.y.z[1] and x.y.z[0] first.

Note: It is up to the user to make sure the kind of data in the value is appropriate. (For example, making sure to put a calendar date into a metadata tag expecting a date.)

#### Examples:

```
addValue($my_id, "data.date.hour[0]", "0700");
addValue($my_id, "data.date.hour[1]", "0800");
addValue($my_id, "control_parms.flags.permissions", "rwx");
addValue($my_id, "data.date.hour[0]", "7:00am"); # overwrites first
example
```

### 6.2.2 attachNote

*attachNote*(*metadata\_id*, *subject*, *body*) returns string

*metadata\_id*: string id of any metadata in the database -- a model metadata, experiment metadata, expstep metadata, or data metadata. This is the 'referenced metadata', eg, the metadata that the note refers to! A note can be attached to any science objects.

*subject*: The note subject, passed as a string.

*body*: The actual note, passed in as a string.

**Returns:** either the database name of the note, or undef on errors. On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg. The name of the note is a string, and can be passed as the *third* argument to the metadataNameSearch function.

**Possible Errors:** Most common: Unable to find the metadata named. Unable to connect to the daemon. Daemon is not connected to the ESSW database.

**Explanation:** ESSW is meant to act as the electronic equivalent of a scientist's notebook. This Perl function can be used to attach text notes to your experiments, models, data or steps. The database will attach ownership and timestamp information to the note, and store it permanently in the database. Note that right now, you will not be able to delete a metadata from ESSW until all the notes that refer to that metadata are deleted.

You must have called `saveToDB()` on the referenced metadata before you can call `attachNote()`. That means that a note can only be attached to a valid database object.

**Examples:**

```
my $notename = attachNote("930302", "Bug Fix", "Im running this to test  
my latest bug fix.") or die "930302 not found";
```

### 6.2.3 beginXMLBld

*beginXMLBld(userid, essw-instance, config-file-path)*

*userid:* your userid. This argument is optional, and can be left out. In those cases, the userid 'essw' will be used. Right now, ESSW is not checking the validity of a userid, so please be careful with this argument to pass a "real" userid. Note that if you need to specify *essw-instance* or *config-file-path*, you will have to specify a userid too, because of the order of the arguments.

*essw-instance:* A symbolic name for the ESSW XML-builder server that you wish to attach to. If this argument is not set, the default *essw-instance* is "PRODUCTION". The current plan is to run at least two XML-builder servers, "TEST", for testing your perl scripts, and "PRODUCTION" for production of permanently stored data products. This *essw-instance* symbolic name corresponds to a section of the config file. If *essw* cannot find a section in the config file with the name you supply, it will return an error.

*config-file-path:* location of the config file to use. If this is not set, *beginXMLBld* will attempt to use the default *essw* configuration file, which is `$ESSWHOME/essw.config`. To use this default, you will have to set the `$ESSWHOME` environment variable.

**Returns:** 1 on success, or undef on failure.

**Possible Errors:** Unable to find the configuration file. `$ESSWHOME` not set. Unable to find the *essw-instance* section of the config file. Unable to find the needed configuration keywords in the config file. *beginXMLBld* needs two config file keywords to be set: `DAEMONPORTNUMBER`, and `DAEMONHOST`. Unable to connect to the daemon. Able to connect, but unable to perform a basic system check.

**Explanation:** *beginXMLBld* starts a session of building XML. This searches for the XMLBld daemon, and connects to it. Information on what XMLBld daemon to connect to is set in the config file.

The first argument to *beginXMLBld* is a *userid*: if you use the Perl API to store metadata, you will want to attach your correct *userid* to it.

The second and third arguments to *beginXMLBld* tell it how to find the config file, and what section of that config file to look in.

Use *endXMLBld* to cleanup and shutdown your connection to the daemon.

### Examples:

```
beginXMLBld ("pete", "TEST"); # Use the "TEST" section of the default
config file, essw.config.

beginXMLBld() # Use ALL defaults: "essw", "PRODUCTION", "essw.config"
beginXMLBld("joe", "PRODUCTION", "/foo/bar/my_special_configfile");
```

## 6.2.4 checkMetadataId

*checkMetadataId(id) returns string*

*id*: string id of any metadata in the database, either metadata about an ExpStep or Data.

**Returns:** the name of the metadata's template. On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Unable to find the metadata named. Also: Unable to connect to the daemon. Daemon is not connected to the ESSW database.

**Explanation:** This function can be used to check for the existence of a piece of metadata, and can also be used to find out the 'template name' of a piece of metadata. This returns undef when the metadata can't be found.

### Examples:

```
$my_template = checkMetadataId("334") or die "Metadata 334 not found";
print "Metadata 334 has a template named $my_template";
```

## 6.2.5 checkXML

*checkXML(metadata\_id) returns string*

*metadata id*: string id of any metadata not yet submitted to the database

**Returns:** either the string \*OK\* if no errors were found in the XML, or undef on errors. In addition, the error message is placed in \$xmlbld::errmsg.

**Possible Errors:** Unable to find the metadata named. Unable to connect to the daemon. Daemon is not connected to the ESSW database. Metadata has already been submitted to the database (at this point it's too late -- right?).

**Explanation:** This method checks that the XML you have built is syntactically correct according to its Template (DTD). This can be used to check for correctness before calling saveToDB().

### Examples:

```
checkXML($my_metadata_id) or die "XML Errors: $xmlbld::errmsg";
```

## 6.2.6 createMetadata

*createMetadata (Template\_name) returns metadata\_id*

*template\_name*: string name of the DTD in ESSW database. If the DTD\_name starts with the string "file:", XMLBld will try to find a file with that name. Otherwise, it will try to find the latest version of a Science metadata Type in the ESSW Database with that name.

**Returns:** a metadata\_id, or undef on failures. The metadata\_id is an internal ID of the current created XML, which is assigned by the server program. You will pass this id to other functions that manipulate this chunk of XML. You shouldn't ever have to use or even look at the value of this id. On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Unable to find the DTD. Unable to connect to the daemon. Daemon is not connected to the ESSW database. Metadata\_id was already previously created.

**Explanation:** createMetadata creates a new empty XML. The DTD\_name specifies the name of the DTD on which to build this XML. You can have many metadatas open simultaneously. Once you have called createMetadata, you can start to build up your XML structure using calls to the addValue() function.

#### Examples:

```
#Initialize some XML metadata based on a Science metadata type called
"personnel", found in the ESSW database:

$my_id = createMetadata ("personnel") or die "Unable to create metadata:
$xmlbld::errmsg";

#Do the same thing, but look for the type in a DTD file:

$my_id = createMetadata ("file:/usr/bob/personnel.dtd") or die "Unable
to create metadata:  $xmlbld::errmsg";
```

### 6.2.7 createExperimentMetadata

*createExperimentMetadata(model\_name)* returns *experiment\_id*;

*model\_name*: string name of the model associated with this run.

**Returns:** an experiment\_id, or undef on failures. The experiment\_id is a unique string id that you can use to refer to this experiment in other perl functions. On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Unable to find the indicated model. Unable to connect to the daemon. Daemon is not connected to the ESSW database.

**Explanation:** An 'experiment' is the running of an earth science model. The link between the experiment and its model is captured from the model name parameter. An experiment may consist of one or more 'experiment steps'. The createExperiment is used to register the existence of a new experiment with the ESSW database. You can call this function before, during, or after the actual running of the experiment.

If you need to attach other useful information to the experiment, use the attachNote() XmlBld function. If you need to attach data lineage information, use the registerExperimentInputs() and registerExperimentOutputs() functions.

The experiment\_id is unique. You can use it, even in another Perl script running at another time, to refer back to this experiment.

You can get a list of all the models in ESSW by running the metadataNameSearch("model") call.

**Examples:**

```
$thisexpid = createExperiment("mymodel");
```

## 6.2.8 createExpStepMetadata

*createExpStepMetadata (experiment\_id, template\_name) returns step\_id*

*experiment\_id*: string id of an experiment

*template\_name*: string name of the step metadata template for the XML to be generated.

**Returns:** a step\_id, or undef on failures. The step\_id is a unique id assigned to the metadata you are about to create. You can pass this id to other functions that manipulate this chunk of XML. On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Unable to find the template named. Other possible errors: No such experiment\_id; Unable to connect to the daemon; Daemon is not connected to the ESSW database; Template\_name is not an 'expstep' template.

**Explanation:** createExpStepMetadata is specially designed to create metadata about the execution of a computer program or application. In ESSW, such an execution is called 'experiment-step', or ExpStep for short. The createExpStepMetadata function expects that you have already created an XML "DTD" template that describes YOUR ExpStep. The createExpStepMetadata function creates a new empty XML representing your ExpStep metadata. The first parameter is the id of the experiment in which this step is running. The step is part of an experiment run, together with other science objects. The second parameter is the name of your DTD template, which must already have been submitted to the ESSW, via the ESSW console program. You can fill-in the empty XML by calling the addValue() function, then call saveToDB() to force the XML to be stored permanently.

**Examples:**

```
Initialize some XML metadata based on a Science metadata for an ExpStep called "fourier_xform", found in the ESSW database:  
$my_id = createExpStepMetadata ($current_exp, "fourier_xform") or die  
"Unable to create expstep metadata: $xmlbld::errmsg";
```

## 6.2.9 closeLogFile

closeLogFile ()

**Returns:** 1

**Possible Errors:** None.

**Explanation:** closeLogFile closes and saves the log file created by the logToFile call. After closeLogFile is called, messages will resume appearing at STDOUT.

## 6.2.10 closeMetadata

closeMetadata (metadata\_id)

*metadata\_id*: string (from the call to createMetadata) id for the metadata the users wish to modify.

**Returns:** The string \*OK\* on success, or undef on errors. Also, on errors, an error message is put in the variable \$xmlbld::errmsg.

**Possible Errors:** Unknown metadata\_id. Not connected to the daemon. Id was already closed previously.

**Explanation:** closeMetadata deletes the XML object corresponding to this metadata\_id. Since this function deletes the in-memory XML, you need to save the XML to a file or to the database before you call this function. There are three functions to retrieve or save the XML: saveToFile, saveToDb, or getMetadataAsString.

**Examples:**

```
closeMetadata ($my_id) or die "Unable to close: $xmlbld::errmsg";
```

### 6.2.11 endXMLBld

endXMLBld ().

**Returns:** undef on failure.

**Possible Errors:** Connection to the daemon may have failed.

**Explanation:** endXMLBld shuts down the session that was started by calling function beginXMLBld or beginRemoteXMLBld. This shuts down the connection, but does not shut down the xmlbld daemon. You can then run other Perl xmlbld sessions to attach to the same daemon.

### 6.2.12 getMetadataAsString

*getMetadataAsString (metadata\_id)*

*metadata\_id*: (from the call to createMetadata) id for the metadata the users wish to read.

**Returns:** returns the entire XML built so far as a Perl scalar value, or undef on errors. Also, on errors, an error message is placed in the \$xmlbld::errmsg variable.

**Possible Errors:** No such metadata\_id.

**Explanation:** getMetadataAsString is the function to get the text XML document from the server. The getMetadataAsString function returns a string, which can be saved to text file or printed on screen.

**Example:**

```
$my_strings = getMetadataAsString($my_id) or die "Unable to get XML: $xmlbld::errmsg";
```

### 6.2.13 esswlink

*esswlink (FROM\_metadata\_id, TO\_metadata\_id, linktype) returns string*

*FROM metadata id*: string id of a metadata in the database

*TO metadata id*: string id of a metadata in the database

*linktype*: Must be one of the following strings: 'input\_to\_experiment', 'input\_to\_expstep', 'expstep\_to\_output', 'experiment\_to\_expstep', 'experiment\_to\_output', 'experiment\_to\_subexperiment', 'data\_to\_data', 'model\_to\_diff'.

**Returns:** either the string \*OK\* on success, or undef on an error. On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Unable to find the metadatas named. Types of the metadata don't match the linktype string used; Invalid link type string used; Unable to connect to the daemon; Daemon is not connected to the ESSW database.

**Explanation:** This function provides the power to link objects together in the ESSW. The allowable ways in which objects can be linked are listed in the linktype above. (You must make sure the types of your metadatas matches the types in the linktype string: NOT IMPLEMENTED). The link function allows relationships between objects in the database. For example, if there are related data, such as a browse image and the full image, they can be linked using the 'data\_to\_data' linktype. This function should be used with care. It is an error to link the same two metadatas with the same linktype more than once.

You must have called saveToDB() on both metadatas before your call esswlink(). Once something is linked, it cannot be removed from the ESSW database until the link is removed first.

**Examples:**

```
esswlink($fullimg, $mybrowseimage, 'data_to_data');
```

## 6.2.14 getFieldNames

*getFieldNames (String table name) returns array*

*Table Name*: valid table names for which the columns are needed.

**Returns:** array of column names for the given table on success or the error message from the database server in case of failure.

**Possible Errors:** The table name might not be correct when you put as an argument.

**Explanation:** This function returns an array of field name list for the given table.

**Example:**

```
# this would return an array containing {"internal_id", "metadata", "typename", "timestamp",  
"typeversion", "objname", "userid"}  
@arrayOfNames = getFieldNames(sciobj);
```

## 6.2.15 logToFile

*logToFile (filename)*

*filename*: string name of the log file.

**Returns:** undef on failure to open the file, so you can call "or die..." on it.

**Possible Errors:** Can't open the file.

**Explanation:** Most perl xmlbld functions print a message every time they are called. You should use the logToFile function to have these messages sent to a file. The message "\*OK\*" is printed on success, or an error message starting with the string "\*ERR\*" is printed. If the logToFile function is not called, the messages will output on STDOUT. Use the closeLogFile to turn off logging to a file, and log to STDOUT again.

**Example:**

```
logToFile ("myXML.log") or die "Unable to open file: $!";
```

## 6.2.16 metadataNameSearch

*metadataNameSearch(templatename, templateversion, name) returns an array of arrayrefs*

templatename: template name to search for. Can be the empty string to do a 'wildcard' search on this field. Put the '%' character in your templatename search string to do a wildcard substring search.

templateversion: template version number to search for. Can be the empty string to do a 'wildcard' search on this field. Pass the string "current" to this argument to get just the **current** version of the template. In most cases, it should be sufficient and correct to search on 'current' as the templateversion. Keep in mind that searching on "" as template version will search on all versions, not just the current version.

name: string name of a metadata to search for. Can be the empty string to do a 'wildcard' search on this field. Put the '%' character in your name search string to do a wildcard substring search. It is possible for two metadatas with different templates, to have the same metadata name. So, in general, if you are searching on 'name', you should usually be searching on 'templatename' as well.

**Returns:** On success, returns a perl "array of references to arrays". The return value is totally identical in structure to that returned by our ESSW perl Query function. The **order of the columns** is guaranteed to be: **id, templatename, templateversion, metadata\_name**.

An example of how to get the id's from this structure is included in the examples below. Undef is returned on errors. If undef is returned, an error message is placed in \$xmlbld::errmsg.

**Possible Errors:** No search results found. Failure to connect to the database. In general, incorrect search strings are considered errors since they will most likely return no search results.

The version argument must be either a number string >= 0, eg, "0" or "2", or the word "current". Any other value is an error.

**Explanation:** metadataNameSearch uses the three search fields to look for metadatas that match. It then returns an array of object "id's" that matched. The array might contain one, or many return values.

Most of the other ESSW Perl API functions take these 'id's' as arguments. Thus this metadataNameSearch function is very useful in translating names of objects in the database into their id's.

For more information on arrays of arrayrefs, and lots of clever tricks, read pages 257 - 264 in "Programming Perl" from O'Reilly press.

**Examples:**

```
my @all_models = metadataNameSearch("model", "", "") or die "search failed: $xmlbld::errmsg";
my @all_avhrr_models = metadataNameSearch("model", "", "avhrr") or die "search failed: $xmlbld::errmsg";
my @just_the_latest_avhrr_model = metadataNameSearch("model", "current", "avhrr") or die "search failed: $xmlbld::errmsg";
my @ALL_metadatas = metadataNameSearch("", "", "");
for $meta(0..$#ALL_metadatas) {
    print "id:      $resultset[$meta][$0] ";
    print " template name: $resultset[$meta][$1] ";
    print " template version: $resultset[$meta][$2] ";
    print " metadata name: $resultset[$meta][$3] ";
    print "\n";
}

my @notes_that_start_with_bob = metadataNameSearch("note", "", "bob%");
```

### 6.2.17 query

*query(string querystr) returns an array of arrayrefs.*

*querystr* : string select query to run. Right now we don't support 'newline' (\n) characters embedded anywhere in the query. The querystr must be a select statement.

**Returns:** the results of the query as an array of arrayrefs, or undef on an error. Each arrayref in the array would represent ONE ROW in the resultset. Then each scalar in the arrayref would represent ONE COLUMN in that row.

On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Any kind of SQL Error. Not a select statement. Unable to connect to the daemon. Daemon is not connected to the ESSW database. Using newline chars in the query.

**Explanation:** This function can be used to send an arbitrary **select** statement to the database. It returns a two-dimensional structure with the results. The names of the columns and types of the columns are not provided. All individual column values are Perl scalars.

For more information on arrays of arrayrefs, and lots of clever tricks, read pages 257 - 264 in "Programming Perl" from O'Reilly press.

**Examples:**

```

my @resultset = query("select Llavhrrfilename from avhrrarchive where
passdate >= 1/1/1999") or die "SQL select error: $xmlbld::errmsg";

# Now here's how to iterate through the resultset.  Another example is
in the metadataNameSearch()
# documentation.
#
# (The $# means to get the length of the array.)
for $i (0 .. $#resultset) {
# The { } means to dereference the pointer. ($resultset[$i] is a
reference)
for $j (0 .. ${$resultset[$i]}) {
print "$resultset[$i][$j]\t";
}
}
print "\n";
}

```

## 6.2.18 query\_hash

*query\_hash* (String querystr) returns hash structure.

*querystr*: string containing the query to run. Right now we don't support 'newline' (\n) characters embedded anywhere in the query. The querystr must be a SELECT statement. No INSERT, UPDATE or DELETE are supported.

**Returns:** If it succeeds, it returns the results of the query in a hash structure. The keywords in the hash structure represent table columns requested through the select statement. The value for each keyword, is an array of strings, indexed by there record in the result set. On failure, returns the error message from database server.

**Possible Errors:** Any kind of SQL Error. Not a select statement. Unable to connect to the daemon. Daemon is not connected to the ESSW database. Using new line chars in the query.

**Explanation:** The difference between this function and the simple query function is that in this case the API, returns a hash object, rather than an array. The hash object can be used to identify the query results by the columns names, rather than using a column index. This approach makes the database access a lot safer. The hash keyword is a column name, and its values is an array of strings. Arrays are used, rather than plain values in order to cover the situation when multiple rows are returned as the query result. For example, if a select statement returns two table rows, this is how the result will look like:

```
My $hashRef = queryHash(Select column1, column2 from myTable);
```

Keyword	Value
Column1	[row1Value, row2Value]
Column2	[row1Vallue, row2Value]

### Examples:

```

# get all records from the SciObj table
my $rset = query_hash("select * from sciobj");

```

```

# now print all the internal Ids from all rows returned
printresults($rset->{internal_id});

# here's how to iterate through the array, each iteration prints the
chosen value for ONE record. The loop counter will max out at the
number of rows retrieved by the query.

sub printresults {
  my @arrayOfColumnValues = @_;
  for $j (0 .. $#arrayOfColumnValues) {
    print "\t$arrayOfColumnValues[$j]\t";
  }
  print "\n\n";
}

```

### 6.2.19 registerExpStepInputs, registerExpStepOutputs

*registerExpStepInputs(expstep\_id, data\_id, data\_id, ...)*  
*registerExpStepOutputs(expstep\_id, data\_id, data\_id, ...)*

expstep\_id: string id of a specific experiment step, as created by createExpStepMetadata.

data\_id: string id of a metadata to associate with this ExpStep. One or more ids are allowed. (...They must be 'data' XML, not 'expstep' XML: this test is currently not implemented.)

**Returns:** the string \*OK\* on success, or Perl undef on failures. Also, on errors, an error message is put in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Incorrect ids. Associating the same data with the same experiment step twice.

**Explanation:** The ESSW tracks the "lineage" of science data. In order to do this, these *registerExpStepXXX* functions are provided to tell the ESSW about the inputs and outputs of an ExpStep. You can call these functions as many times as you want with the same ExpStep. For example, a data can be the Output of one step, and also be the Input of many other steps.

To protect against possible errors in the XML, you must call saveToDB() on all the metadatas listed (including the ExpStep metadata) before you call these functions. You can associate the same data XML with many different experiment steps.

#### Examples:

```

registerExpStepOutputs($myStepId, $outputid1);
registerExpStepInputs($myStepId, $in1, $in2, $in3, $in4, $in5)
registerExpStepOutputs($myStepId, $output2, $output3);

```

### 6.2.20 registerExperimentInputs, registerExperimentOutputs

*registerExperimentInputs(Experiment\_id, data\_id, data\_id, ...)*  
*registerExperimentOutputs(Experiment\_id, data\_id, data\_id, ...)*

Experiment\_id: string id of a specific experiment step, as created by createExperimentMetadata.

data\_id: string id of a metadata to associate with this Experiment. One or more ids are allowed. They must be 'data' XML, not 'Experiment' XML.

**Returns:** the string \*OK\* on success, or Perl undef on failures. Also, on errors, an error message is put in the variable \$xmlbld::errmsg.

**Possible Errors:** Most common: Incorrect ids or ids. Associating the same data with the same experiment twice.

**Explanation:** An Experiment can be thought of as a 'black box' with a bunch of ExpSteps hidden in it. These functions let you track the inputs and outputs of the entire Experiment. Many data XMLs will be registered as inputs or outputs of *both* an ExpStep and the Experiment. For example, the outputs of the Experiment are actually always outputs of some ExpSteps inside that Experiment. Likewise, the inputs to the Experiment are really inputs to ExpSteps inside that Experiment.

You can call these functions as many times as you want with the same Experiment. You can associate the same data XML with many different experiments. For example, a data can be the Output of one experiment, and also be the Input of many other experiments.

To protect against possible errors in the XML, you must call saveToDB() on all the metadatas listed before you call these functions. You can associate the same data XML with many different experiments. An XML can be associated with any combination of both experiments and Experiment steps.

**Examples:**

```
registerExperimentOutputs($exp, $outputid1);
registerExperimentInputs($exp, $in1, $in4)
registerExperimentOutputs($exp, $output2, $output7);
```

### 6.2.21 saveToFile

*saveToFile(id, filename)*

*id:* string (from the call to createMetadata) id for the metadata the users wish to save. This is actually the ID of an XML document from the xmlbld Java daemon memory space.

*filename:* path location where to save the XML. Note: this is a path on the machine where the daemon runs, and is relative the daemon's current directory!

**Returns:** "OK" on success, or undef on errors. Also, on errors, an error message is placed in the \$xmlbld::errmsg variable.

**Explanation:** saveToFile saves the XML object to the server's disk.

**Possible Errors:** Incorrect path. No write permissions.

**Example:**

```
saveToFile ($my_id, "mydata.xml") or die "Unable to save XML to file:
$xmlbld::errmsg";
```

### 6.2.22 saveToDB

*saveToDB(id, name\_to\_use, ingest, filename1, filename2, filename3, ....)*

*id:* string (from the call to createMetadata) id for the metadata the users wish to save.

*name\_to\_use*: name to use for this object in the ESSW database. If this name is not unique in the database, the ESSW will assign a name by attaching a *<integer>* to the name to generate a unique name. Pass the empty string "" as the *name\_to\_use* if you want the ESSW to assign a name for your data object.

*ingest*: a boolean. Indicates whether to ingest data files or not. True (eg, "1") means to copy data files into internal database-managed disk space. False ("", or undef) means to just track the data's location outside the database, but don't copy it anywhere. NOTE: this feature is not yet implemented.

*filename(s)*: string path of data files to associate with this metadata. If *ingest* is true (anything other than "" or undef), then the data will be copied into the database. If *ingest* is false (undef or ""), then the data files won't be moved anywhere. You can list as many files as you want. NOTE: this feature is not yet implemented.

**Returns:** The unique name assigned to that object in the database, or undef on errors. Note that you should not assume that *name\_to\_use* was the name actually assigned in the database. Also, on errors, an error message is placed in the `$xmlbld::errmsg` variable.

**Explanation:** `saveToDB` saves the XML object of the server runtime memory to the server's database. The "ingest" argument indicates whether to ingest data files or not. The number of filenames can be varied.

**Possible Errors:** Not connected to the ESSW database. Incorrect id. Data filename(s) don't exist. Passing wrong number of parameters

#### Examples:

```
$essw_id = saveToDB ($my_id, "Tuesdays_data", "", "mydata1", "mydata2")
or die "Couldn't save data to ESSW: $xmlbld::errmsg";

# Store metadata, but no actual science files. Ask ESSW to assign
# the name:
$id_array[$i] = saveToDB ($my_id, "", "") or die "Couldn't save data to
ESSW: $xmlbld::errmsg";

# Filenames are in perl scalar variables. DO store the actual data in
# the database. Ask ESSW to assign the name
$resultname = saveToDB($metadata[$i], "", "1",
$run_2001_ctrl_file,$run_2001_cmd_file, $run_2001_output_file) or print
"error in saveToDB: $xmlbld::errmsg";
```

### 6.2.23 setErrorHandlingMode

*setErrorHandlingMode(mode\_name)* returns string

*mode\_name*: string name of mode to use. Only the following modes are valid: "abort", "continue". (other modes can be added as the need arises)

**Returns:** either the string \*OK\* if the function worked, or undef on errors. If undef is returned, an error message is placed in `$xmlbld::errmsg`.

**Possible Errors:** Incorrect mode name used.

**Explanation:** `setErrorHandlingMode` tells XMLBLD what to do when errors occur in building a chunk of XML, in particular, what to do when an `addValue` call fails.

In "abort" mode, any metadata being built is written to local files. (The filename used is 'metadata\_<id>.xml, where <id> is the id of the metadata. Then the perl script is exited. This allows the user to at least retain metadata that has been collected so far. In "continue" mode, errors in XMLBLD routines are written to the \$xmlbld::errmsg variable, but the script continues to run.

"abort" mode is the default. Note that the getMetadataAsString function can be used at any time to get a partially built XML if the user wishes to write her own abort mechanism.

**Examples:**

```
setErrorHandlingMode("continue") or die "Unable to change error modes:
$xmlbld::errmsg";
```

### 6.2.24 shutdownServer

*shutdownServer ()*

**Returns:** 1 on success or undef on errors.

**Explanation:** shutdownServer shuts down the server daemon, and then closes the connection. This call can only be used to shutdown a local daemon. Remotely connected perl scripts cannot shut down the daemon.

**Possible Errors:** Attempting to shutdown when not connected to the daemon. Attempting to shutdown from a remote connection.

### 6.2.25 sysCheck

*sysCheck () returns array*

**Returns:** returns an array of useful information or null if there was an error.

retval[0] : the version of the xmlbld daemon you are connecting to.

retval[1] : the port number the xmlbld daemon is listening on.

retval[2] : the full name and path of the log file the xmlbld daemon is using for logging informational and debugging messages.

retval[3] : the log mode of the xmlbld daemon. Currently only two modes are available: ON - when all information is logged, or OFF - when no information is logged.

retval[4] : the user id of the xmlbld daemon process.

retval[5] : the status of the database connection the xmlbld daemon is using. It can be a valid connection ("\*OK\*" is displayed). If there are problems connecting to the database, "\*FAIL\*" is displayed.

retval[6] : the host name where the database server is running.

retval[7] : the host name of the machine where the daemon is running.

**Possible Errors:** Unable to connect to the daemon.

**Explanation:** This function returns configuration information about the current state of the daemon that the invoking program connects to.

**Example:**

```
@retval = syscheck() or die "XML Errors:$xmlbld::errmsg";  
print "I am running version $retval[0]";
```

## 6.2.26 setVerbose

*setVerbose (String path)*

*path:* path name and file name of a valid logfile where the information should be written.

**Returns:** OK on success, null on failure.

**Possible Errors:** Daemon does not have permissions to write to the log file, or create the log file. XML daemon is not running.

**Explanation:** This function turns daemon logging on or changes the current log file. If logging is not on - it's turned on and directed to the specified file. It can send the logging information to a new log file - whose path and file name are specified in the call.

**Example:**

```
setVerbose("usr/local/esswlog");  
setVerbose();
```

## 6.3 Prototype Lineage Perl API and Commands

This is a temporary placeholder for prototyping lineage Perl API and commands. This section will be moved into section 6.2 in the final version of the manual. There are two subsections, one subsection for Perl APIs that can be called from other Perl programs and one subsection for Perl scripts that can be run by users.

In this section and the next section, we will use the following experiment graph as an example:



### 6.3.1 Prototype Lineage Perl API

The Perl modules containing the lineage information are SciObject.pm, Compare.pm, LineageCompare.pm, lineage.pm in directory `#{ESSWHOME}/examples/lineage`. A science object is wrapped into SciObject type object. Each SciObject object has 4 attributes: id, name, type, version. In our examples, we will only show its id, like (2, ...). Compare.pm implements the compare algorithm to diff two objects. LineageCompare.pm implements the diff algorithm for two lineages. The module lineage.pm exports the following functions.

#### 6.3.1.1 getStepForData

*getStepForData (Integer obj\_id) returns integer*

*obj\_id*: internal object id of a data object

**Returns:** If it succeeds, returns the internal id of the experiment step that produced the data object *obj\_id*. If it fails, returns null.

**Possible Errors:** *obj\_id* is not in database or is not belong a data object.

**Explanation:** This function traverses the experiment graph backwards to find the step id of the step that produced the desired data.

**Example:**

```
# Refer to the diagram at the beginning of this section
getStepForData(9) returns 8
```

#### 6.3.1.2 getInputsForStep

*getInputsForStep (Integer step\_id) returns a hash table data structure*

*step\_id*: internal object id of an experiment step

**Returns:** If it succeeds, returns a reference to a hashtable where keys are data type names and values are a references to an array of input SciObject structures, that served as input to the specified step. If fails, it returns null.

**Possible Errors:** *step\_id* is not in database or does not represent an experiment object.

**Explanation:** This function traverses the experiment graph backwards and identifies all input objects to the specified *step\_id*. The returning hash table will contain the input object types as keywords and an array of SciObject structures as value. The SciObject structures identify the specific objects with more than their Ids. They provide the name, type and version for the objects.

**Example:**

```
# Refer to the diagram at the beginning of this section

# Let's assume that step 8 has two inputs of type D3 and D4, with the
  Ids 6 and 7

getInputsForStep(9)

returns a hashtable with two keys D3 and D4 where
value{D3}=[(6, ...)]
value{D4}=[(7, ...)].
```

### 6.3.1.3 getInputsOfDepthForData

*getInputsOfDepthForData (Array dataObjects, Integer depth) returns array of arrays*

*dataObjects*: an array of data object ids

*depth*: the number of steps that will be traversed back from the indicated outputs

**Returns:** If it succeeds, returns an array of arrays, the first index equals the depth of data object to outputs. If fails, it returns null.

**Possible Errors:** no objects with the indicated ids are in database, depth < 0 or objects in the input array are not data objects.

**Explanation:** This function traverses the experiment graph backwards to find all input objects that produced the specified outputs. The input objects will be inserted into an array according their distances from the indicated outputs.

**Example:**

```
# Refer to the diagram at the beginning of this section

# Assuming that data object 9 originated from objects 6 and 7, and
  object 6 originated from objects 1, and object 7 originated from object
  2

getInputsOfDepthForData([9],2)
will return
[
  [(9, ...)],
  [(6, ...), (7, ..)],
  [(1, ..), (2, ..)]
].
```

### 6.3.1.4 getOutputsOfDepthForData

*getOutputsOfDepthForData (Array inputs, Integer depth) returns array of arrays*

*inputs*: an array of data object ids

*depth*: the number of steps that will be traversed forward from the specified Ids.

**Returns:** If it succeeds, returns an array of arrays where the first index equals the depth of the data objects to the specified inputs. If fails, it returns null.

**Possible Errors:** no objects with indicated ids are in the database, depth < 0 or the objects in the input array are not data objects.

**Explanation:** This function traverses the experiment graph forward to find all output objects that were produced from the specified inputs. The outputs are inserted into arrays according their distances from inputs.

**Example:**

```
# Refer to the diagram at the beginning of this section
# Assuming that object 1,2 produced objects 6 and 7 - and objects 6 and
7 produced object 9
getOutputsOfDepthForData([1,2], 2)
returns
[
  [(1, ...), (2, ...)],
  [(6, ...), (7, ...)],
  [(9, ...)]
].
```

### 6.3.1.5 getStepsForwardOfDepthForData

*getStepsForwardOfDepthForData (Array inputs, Integer depth) returns array of arrays*

*inputs:* an array of data object ids

*depth:* the number of steps that will be traversed forward from the specified data objects

**Returns:** If it succeeds, returns an array of arrays where the first index equals the depth of the step objects from the specified inputs.

**Possible Errors:** indicated inputs are not in the database, depth < 0 or the object in the array are not data objects.

**Explanation:** This function traverses the experiment graph forward to find all steps that are affected by the specified data objects, then inserts them into arrays according their distances from the indicated inputs.

**Example:**

```

# Refer to the diagram at the beginning of this section

# Assuming that object 1 and 2 served as inputs to steps 4 and 5 and
objects 6 and 7 (produced by steps 4 and 5) served as input to step 8

getStepsForwardOfDepthForData([1,2], 2)
returns
[
    [(4, ...), (5, ...)],
    [(8, ...)]
].

```

### 6.3.1.6 getExperimentForData

*getExperimentForData (Integer obj\_id)*

*obj\_id*: the internal id of a data object

**Returns:** If it succeeds, returns an array of size 2 where the first element is an array of experiment objects that use specified object id as an input, and the second element is the experiment object that produced the specified object id. If it fails, returns null.

**Possible Errors:** specified object id is not in database, or the object specified is not a data object.

**Explanation:** This function identifies the experiment that produced the desired data object, and the experiment that uses the desired data object as an input.

**Example:**

```

# Refer to the diagram at the beginning of this section

getExperimentForData(1)
returns
[[ (3, ...) ], []], meaning that there are no experiments that produced
object 1
getExperimentForData(6)
returns
[[], [(3, ...) ]], meaning that there are no experiments that use object 6
as an input.

```

### 6.3.1.7 getExperimentForStep

*getExperimentForStep (Integer step\_id)*

*step\_id*: the internal id of an experiment step

**Returns:** If it succeeds, returns an array of experiment objects that have the specified step\_id as part of their processing sequence. If it fails, returns null.

**Possible Errors:** step\_id is not in the database or the object with the specified id is not an experiment step.

**Explanation:** This function finds all the experiments that have the experiment step with the specified id as part of their processing sequence.

**Example:**

```
# Refer to the diagram at the beginning of this section
getExperimentForStep(4)
returns
[(3, ...)].
```

### 6.3.1.8 getExpObjects

*getExpObjects (int object\_id) returns a hash data structure (hash of arrays)*

*object\_id*: id of a valid experiment

**Returns:** If it succeeds, returns a reference to hash, if it fails it returns an error message.

**Possible Errors:** object\_id is not in the database or the object with the specified id is not an experiment.

**Explanation:** This function finds objects directly associated with an experiment. The associated objects are: experiment steps, inputs and outputs. The hash data structure returned by this function has as keyword the object types (inputs, outputs and steps) and the values are arrays of object ids of each specified type. For example:

```
$rset = {
  inputs → [12 13 14],
  steps → [
    {step_id → 15, input → 12, output → 24},
    {step_id → 18, input → 13, output → 24},
    {step_id → 21, input → 14, output → 25},
  ],
  outputs → [24, 25],
};
```

NOTE: for the step keyword, the value is an array of hashes that contain the following keywords: step\_id, input, output. The corresponding values are arrays of object ids.

**Example:**

```

# Refer to the diagram at the beginning of this section
$reset = getExpObjects(3);

returns
{
    inputs -> [1,2]
    steps  -> [
        {step_id -> [4]
          input  -> [1]
          output -> [6]
        },
        {step_id -> [5]
          input  -> [2]
          output -> [7]
        },
        {step_id -> [8]
          input  -> [6, 7]
          output -> [9]
        }
    ]

    outputs -> [9]
}

#Following is the code to print out the hash of arrays

print "Inputs to Experiment : ";
for $j (0 .. ${$reset->{inputs}}) {
    print "$reset->{inputs}[$j][0] ";
}

print "\Outputs from Experiment : ";
for $j (0 .. ${$reset->{outputs}}) {
    print "$reset->{outputs}[$j][0] ";
}

print "\Inputs to Step:  \n";
for $i (0.. ${$reset->{steps}} ) {
    print "Step ID: @{$reset->{steps}}[$i]->{step_id} ";
    print "  Input: ";
    for $k (0.. ${@{$reset->{steps}}[$i]->{input}} ) {
        print "@{$reset->{steps}}[$i]->{input}[$k][0] ";
    }
    print "  Output: ";
    for $k (0.. ${@{$reset->{steps}}[$i]->{output}} ) {
        print "@{$reset->{steps}}[$i]->{output}[$k][0] ";
    }
}

```

### 6.3.1.9 getLineage

*getLineage(Int. start\_id, Int. direction, Int. depth, String object\_type, String link\_type)*

*start\_id*: the internal\_id of a SciObject from which to begin the lineage search.

*direction*: specifies a forward or reverse direction.

*depth*: specifies how far to search. A depth of -1 means search as far as possible.

*object\_type*: specifies the type of object to return. Valid values would be any valid SciObject derived type in ESSW or "all" which implies all objects that are found.

*link\_type*: specifies a comma-delimited string of any of the valid link types, e.g. "input\_to\_experiment,input\_to\_expstep" or "all", which specifies that all link\_types found in lineage path should be returned .

**Returns:** If it succeeds, returns a reference to a hash of all the objects found. The columnized data associated with each object is included in the hash. If it fails it returns an error message.

**Possible Errors:** Object with internal\_id equals to start\_id is not in the database; invalid object\_type or invalid link\_type string.

**Explanation:** This function finds objects that are in the lineage path of an object with internal\_id matching that of the start\_id. The direction of the search is designated by the constant \$REVERSE\_DIRECTION or \$FORWARD\_DIRECTION which translates to 0 and 1 respectively in "lineage.pm". The depth of the search is designated by a number which represents the level or depth of the search starting from the object with internal\_id matching that of start\_id; "-1" implies, to search as far as possible in the designated direction.

Usage is as follows:

```
$hashOfHashRef = getLineage(42, 1, 2, 'all', 'all');
@ArrayKeys = keys (%$hashOfHashRef);
foreach $key (@ArrayKeys) {
    print "$key\n";
    $InnerHashRef = $hashOfHashRef->{$key};
    @InnerKeys = keys (%$InnerHashRef);
    foreach $Column (@InnerKeys) {
        print "$Column=>{$InnerHashRef->{$Column}}";
    }
}
```

Sample result is as follows:

```
46
typename=>step2
internal_id=>46
os_type=>LINUX
timestamp=>2000-04-12:17:09:49.0
cpu_time=>200.80
typeversion=>0
.....
48
typename=>data3
internal_id=>48
objname=>data3_b.49
os_type=>Solaris 2.6
....
.....
```

## 6.3.2

### 6.3.3 Perl Lineage Scripts

The following are Perl scripts can be used to extract lineage information from the ESSW schema.

#### 6.3.3.1 **getExperimentForData.pl x**

x: the internal id of a data object

**Output:** experiment ids that use x in their inputs and the experiment id that has x as an output.

**Possible Errors:** x is not in database.

**Explanation:** This command uses the function `getExperimentOfObject` described in the previous subsection.

#### 6.3.3.2 **getExperimentForStep.pl x**

x: the internal id of an experiment step object

**Output:** experiment id that includes x in its steps.

**Possible Errors:** x is not in database.

**Explanation:** This command uses the function `getExperimentOfStepObject` described in the previous subsection.

#### 6.3.3.3 **getInputsOfDepthForData.pl x y**

x: the internal id of a data object

y: a positive integer

**Output:** print out all data objects that produced the specified internal id, within distance  $y$  of the specified object.

**Possible Errors:**  $x$  is not in database, or  $y < 0$ .

**Explanation:** This command uses the function `getInputObjectsOfDepth` described in the previous subsection.

#### 6.3.3.4 `getOutputsOfDepthForData.pl x y`

$x$ : the internal id of a data object

$y$ : a positive integer

**Output:** print all data object ids forward from  $x$  within distance of  $y$ . These are objects that are produced from the specified Id, as result of various experiment steps.

**Possible Errors:**  $x$  is not in database, or  $y < 0$ .

**Explanation:** This command uses the function `getOutputObjectsOfDepth` described in the previous subsection.

#### 6.3.3.5 `getStepsForwardOfDepthForData.pl x y`

$x$ : the internal id of a data object

$y$ : a positive integer

**Output:** print all experiment steps ids forward from  $x$  within distance of  $y$ .

**Possible Errors:**  $x$  is not in database, or  $y < 0$ .

**Explanation:** This command uses the function `getStepsFromObjectsOfDepth` described in the previous subsection.

#### 6.3.3.6 `cmp.pl x y`

$x$ : internal id of an object

$y$ : internal id of an object

**Output:** print the differences in graph topology and object types found by traversing backwards the graphs from which  $x$  respectively  $y$  take part.

**Possible Errors:**  $x$  or  $y$  are not in database.

**Explanation:** This command traverses backwards from both lineage graphs that  $x$  and  $y$  are in, comparing all data objects and experiment steps and printing out the comparison results. This script is useful in trying to assess why two outputs, from the same experiment are different. It should be used mainly on graphs with the same topology – otherwise the traversal will stop because of the difference in graph topologies.

### 6.3.3.7 getLineage.pl argv0 argv1 argv2 argv3 argv4

6.3.3.7.1.1.1 argv0: start\_id

6.3.3.7.1.1.2 argv1: direction

6.3.3.7.1.1.3 argv2: depth

6.3.3.7.1.1.4 argv3: object\_type

6.3.3.7.1.1.5 argv4 :link\_type

**Output:** Prints a list of the internal\_ids followed by all the columnized data that is associated with each object with that internal\_id.

**Possible Errors:** start\_id not in database or object\_type or link\_type is invalid

**Explanation:** This function finds objects that are in the lineage path of an object with internal\_id matching that of the start\_id. Valid direction is either 0 (for backward search ) or 1 (for forward search). The depth of the search is designated by a number which represents the level or depth of the search starting from the object with internal\_id matching that of start\_id; "-1" implies, to search as far as possible in the designated direction.

## 6.4 Metadata Standards

Metadata standards is a feature that is added to essw to facilitate the reuse and searching for DTDs and tags based on criteria such as keywords, controlled-vocabulary terms, which are domain related terms that are associated with DTDs.

The following must be done to facilitate metadata standards support.

1. Run new SQL script essw.sql
2. Run essw script from the essw/bin directory: <essw -a notebook\_setup.essw &>
3. Run essw script from essw/bin directory: <essw -a standards\_setup.essw &>
4. Run the SQL script <standards\_setup.sql > against the database
5. DTDs must be written in a specified format with the description fields of the root element and additional element formatted as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT aviris (sensor_name, av_run_start_date, av_run_start_time, av_run_end_date) >
  <!ATTLIST aviris dbtype CDATA #FIXED "lvvarchar"
    description CDATA #FIXED "
      [CONTRLD_VOCAB_IDS] 1,2
      [FULL_ROOTELEM_NAME] Airborne Visible InfraRed Imaging Spectrometer
      [DESCRIPTION] This EXAMPLE DTD provides information about an AVIRIS scene.
      [CREATOR] Tom Painter, UCSB/Bren
      [RELATED_DOCS] AVIRIS Data Tape Documentation, JPL
```

```

[COMMENTS] Place comments here
">

<!ELEMENT sensor_name (#PCDATA)>
  <!ATTLIST sensor_name dbtype CDATA #FIXED "varchar(25)"
  description CDATA #FIXED "
    [FULL_ELEMENT_NAME] Sensor Name
    [DEFINITION] The name of the sensor used to acquire data.
    [SYNONYMS] Spectrometer Name
    [KEYWORDS] sensor, instrument, remote sensing, imagery
    [VALUE_DOMAIN] free text
    [FORMAT] Place format info here
    [EXAMPLE] aviris
    [RELATED_DOCS] Place information about related docs here.
    [COMMENTS] Place comments here.
  ">

```

### 6.4.1 getDTDForVocabTerm

*getDTDForVocabTerm(int termId) returns an array of arrayrefs.*

6.4.1.1.1 *termId*: Id for the vocabulary term of interest

**Returns:** This function returns the fields that constitute the primary key for dtds that are associated with the supplied vocabulary term id. The only argument that is taken is the vocabterm\_id that is associated with the DTD. This function returns an array of arrayrefs. Each arrayref in the array represents one row in the result set and each scalar within each arrayref represents one column in the row. In this case the columns returned are the foreign keys from the elemset table. Since these columns refer to the primary keys in the ElemSet table: name, and version -- they represent each unique DTD that is associated with the supplied vocabterm\_id.

On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Any kind of SQL Error; Unable to connect to the daemon; daemon is not connected to the ESSW database.

**Explanation:** This function can be used to retrieve DTDs that are associated with a vocabulary term of interest.

**Examples:**

```

my @resultset = getDTDForVocabTerm(1);
# Now here's how to iterate through the resultset.
# Two columns would be in this resultset - elemsetname, and
elemsetversion

for $i (0 .. $#resultset) {
  for $j (0 .. ${#resultset[$i]}) {
    print "$resultset[$i][$j]\t";
  }
}
print "\n";
}

```

## 6.4.2 getTagsForKeyword

*getTagsForKeyword(String keyword) returns an array of arrayrefs.*

6.4.2.1.1 *keyword*: keyword of interest

**Returns:** This function returns the fields that constitute the primary key for tags that are associated with the supplied keyword. The only argument that is taken is the keyword of interest. This function returns an array of arrayrefs. Each arrayref in the array represents one row in the result set and each scalar within each arrayref represents one column in the row. In this case the columns returned are the foreign keys from the elemdecl table. Since these columns refer to the primary keys: elemname, elemsetname, and elemsetversion -- they represent each unique tag that is associated with the supplied keyword.

On errors, undef is returned, and an error message can be found in the variable \$xmlbld::errmsg.

**Possible Errors:** Any kind of SQL Error; Unable to connect to the daemon; daemon is not connected to the ESSW database.

**Explanation:** This function can be used to retrieve tags that are associated with a particular keyword.

### Examples:

```
my @resultset = getTagsForKeyword("IVIRIS");
# Now here's how to iterate through the resultset.
# Three columns would be in this resultset - elemname, elemsetname, and
# elemsetversion
for $i (0 .. $#resultset) {
    for $j (0 .. ${$resultset[$i]}) {
        print "$resultset[$i][$j]\t";
    }
    print "\n";
}
```

## 6.4.3 getTagsForDTD

*getTagsForDTD(String dtdname) returns an array of arrayrefs.*

6.4.3.1.1 *dtdname*: Name of the DTD of interest

**Returns:** This function returns the fields that uniquely identify each tag that is associated with a particular DTD. The only argument that is taken is the dtd name of interest. This function returns an array of arrayrefs. Each arrayref in the array represents one row in the result set and each scalar within each arrayref represents one column in the row. In this case the columns returned





\*\*\*\*\*THE UML CLASS AND COMPONENT DIAGRAMS ARE AVAILABLE\*\*\*\*\*

#### Milestone 1B

3. UML diagrams that facilitate a better understanding of the programming model were generated in HTML. The HTML files and the directory structure can be found at `essw/ESSW_ROSE` and `frames.html` would be the starting point to navigate.

\*\*\*\*\*NEW LINEAGE PERL APIs and UTILITY SCRIPTS ADDED\*\*\*\*\*

#### Milestone 1A

1. Lineage walking allows user to walk along the lineage tree forwards and backwards. We now have the following APIs implemented:

`getInputsForStep`, `getInputsOfDepthForData`, `getOutputsOfDepthForData`,  
`getStepsForwardOfDepthForData`, `getExperimentForData`, `getExperimentForStep`

2. Lineage comparison based on columnized tags shows differences between two lineage trees, the API is:

`cmpLineage`

3. Utility scripts for lineage tree walking and comparison were added:

`getInputsOfDepthForData.pl`, `getOutputsOfDepthForData.pl`,  
`getStepsForwardOfDepthForData.pl`, `getExperimentForData.pl`,  
`getExperimentForStep.pl`, `cmp.pl`

-----  
!!!!!!! These are the changes that make up release 0.8 !!!!!!!!  
!!!!!!! since release 0.7.1: !!!!!!!!  
-----

\*\*\*\*\*CHANGES TO THE DATABASE SCHEMA\*\*\*\*\*

Added a new linktype ("`model_to_dif`") and linkid 8. This changed the `EsswLinkType` table, and the `LinkTypeString` function.

\*\*\*\*\*GENERAL NON-VISIBLE CHANGES TO THE CODE\*\*\*\*\*

Moved `xml4j.jar` and `infxjdbc.jar` and `essw.jar` into a new `essw/lib` directory. This means that we can more easily control which version of the libraries we're using, and in addition the user no longer has to set the `CLASSPATH` environment variable!

\*\*\*\*\*CHANGES TO THE PERL API AND XMLBLD DAEMON\*\*\*\*\*

Added the `setVerbose` API () function, that turns logging on and off.

Added the `sysCheck()` function, that gives daemon information.

Bug Fix: Got rid of all calls to 'die' in `xmlbld.pm`, except those called when in the "abort" mode.

Default config file that the daemon and `xmlbld.pm` looks for on unix is now `$ESSWHOME/essw.config`

\$ESSWHOME environment variable MUST be set, or the daemon won't start.

Added -e option to set the ESSW Config file name, when starting xmlbld. (There are now 3 ways to find config file: \$ESSWHOME/essw.config \$ESSW\_CONFIG\_FILE, or -e <filename>) See the User's Manual for details.

Also added -s <instancename> to command line options.

beginXmlBld() now does a simple syscheck and prints info to its log.

Must now start the daemon by hand.

xmlbld daemon prints useful messages on startup.

No longer need CLASSPATH to find java classes -- its now embedded in the command!

Added reading the \$ESSW\_CONFIG\_FILE\_SECTION environment variable if it exists.

Added sections to the config file.

Added 2 new parameters to beginXMLBld API function. Its now beginXmlBld(userid, section, path)

Got rid of beginRemoteXmlBld() perl functions.

\*\*\*\*\*CHANGES TO THE PERL UTILITIES:\*\*\*\*\*

There are now the following perl utility programs in \$ESSWHOME/bin/

loggingon.pl Turns on logging in the xmlbld daemon.

loggingoff.pl Turns off logging in the xmlbld daemon.

shutdown.pl Shuts down the xmlbld daemon

syscheck.pl Prints out useful info on the status of the daemon

getDTDForVocabTerm Prints the DTDs that are associated with a control vocabulary term

getTagsForKeyword Prints the tags that are associated with a particular keyword

getTagsForDTD Prints the tags that are elements of a DTD

shutdown.pl only works if the DAEMONHOSTNAME is set to "localhost"

See the User's Manual for details on how to use them.

\*\*\*\*\*CHANGES TO THE CONSOLE\*\*\*\*\*

Added the setverbose command.

Unix: Default config file is \$ESSWHOME/essw.config

On unix: \$ESSWHOME MUST be set, or its an error.

Added -e option to set the ESSW Config file name, when starting

console (There are now 3 ways to find config file:

\$ESSWHOME/essw.config, \$ESSW\_CONFIG\_FILE, or -e <filename>. See the User's Manual for details on how to use these.)

Console ignores the OPENDBCONNECTIONS and MAXDBCONNECTIONS config file keywords and always uses 1 for both.

Console now prints essw code version and databasename on startup.

No longer need CLASSPATH to find java classes -- its now embedded in the command. You still have to put java1.2/bin in your \$PATH.

Added -s <sectionname> to console command line.

Added reading the \$ESSW\_CONFIG\_FILE\_SECTION environment variable if it exists.

To run the console, you will need to

- 1) have \$ESSWHOME set properly.
- 2) have java1.2/bin in your path
- 3) have ESSWHOME/bin in your path.
- 4) have your config file setup properly.

Here's an example:

```
mars> setenv ESSWHOME /home/jimduff/essw
mars> setenv PATH $ESSWHOME/bin:/usr/java1.2/bin:$PATH
mars> essw
ESSW 0.7.1 configuring from /home/jimduff/essw/essw.config PRODUCTION instance.
Connected to database essw.
```

\*\*\*\*\*CHANGES TO THE ESSW CONFIG FILE\*\*\*\*\*

The config file currently uses the following keywords. Please see the ESSW User's Manual for details on them.

```
PASSWORD=mysecret
USERNAME=dbuser1
DATABASEHOST=foo.bar.baz.edu
DATABASEPORT=5555
DATABASESERVER=my_informix_servername
DATABASENAME=essw_production_db
OPENDBCONNECTIONS=5
MAXDBCONNECTIONS=10
MESSAGELOG="/tmp/esswlog"
CONSOLELOG=consolelog
# Note that the quotes can be used as well:
XMLBLDLOG="production_xmlbldlog"
UNIT="production"
DEBUGSTATUS="NoDebug"
DAEMONPORTNUMBER=8233
DAEMONHOST=foo.bar.baz.edu
```

Optional keywords, not normally needed:

```
LOB_MEMORY_CACHE=65535
DATABASEDRIVER=com.informix.jdbc.IfxDriver
DATABASEURL="jdbc:informix-
sql://tempest.ucsb.edu:5555/essw:INFORMIXSERVER=myserver"
PROTOCOL=informix-sqli
```

If problems are encountered during initialization they will be logged on stdout and in a file /tmp/ESSW\_ERRORS.LOG.<userid>

\*\*\*\*\*CHANGES TO THE EXAMPLES (in essw/examples/)\*\*\*\*\*

Added essw/examples/lineage, with prototype lineage-tracking perl programs.

Added essw/example/dif, with the original dtd and "corrected" dtd. The corrected dtd has a few small typo's fixed.

All the functions have been updated to have this line at the beginning:

```
use lib $ENV{ESSWHOME} . "/perl";
```

which lets the program find the xmlbld.pm API module. ALL Perl programs must now include this line at the beginning.

\*\*\*\*\*CHANGES TO THE NOTEBOOK AND SERVLET ARCHTECTURE\*\*\*\*\*

None.

\*\*\*\*\*CHANGES/ADDITIONS TO THE PREDEFINED TEMPLATES\*\*\*\*\*

There are three predefined templates: model, experiment, note.

They have not changed since the last release.

\*\*\*\*\*CHANGES TO THE PERL TEST FUNCTIONS (in essw/test)\*\*\*\*\*

The perl test functions are used to test the functionality of the perl api.

All the functions have been updated to have this line at the beginning:

```
use lib $ENV{ESSWHOME} . "/perl";
```

which lets the program find the xmlbld.pm API module. ALL Perl programs must now include this line at the beginning.

Added setverbose.pl to test the setVerbose() perl function.

Added syscheck.pl to test the sysCheck() perl function.

Added readconfig.pl, which tests the readconfig function.

-----  
!!!!!!!!!! The following changes apply to release 1.0 !!!!!!!!!!  
-----

\*\*\*\*\*ADDITION OF METADATA STANDARDS SUPPORT\*\*\*\*\*

The database schema is modified to support metadata standards features. Changes include the addition of six new tables – ElemsetMetadata, CtrlVocabTerm, CtrlVocabulary, CtrlVocabTermStruct, Elemsetterm, and ElemDeclMetadata.

All new tables except for ElemSetMetadata and ElemDeclMetadata are added by running the new SQL script essw.sql. ElemSetMetadata and ElemDeclMetadata are added as DTDs by running the script , standards\_setup.essw and standards\_setup.sql, in that order, after the script notebook\_setup.essw is run against the database that is created by running of the script essw.sql.

The following must be done to facilitate metadata standards support.

1. Run new SQL script essw.sql
2. Run the essw command <essw -a notebook\_setup.essw &>
3. Run the essw command <essw -a standards\_setup.essw &>
4. Run the SQL script <standards\_setup.sql > against the database

