EUPS User's Guide

User's Guide

For an abridged version (but not a replacement) of this guide see The Impatient's Guide to DESDM EUPS installation, which contains specific information about installing on OSX (10.7-10.11).

This is the user manual for DESDM's new package management system. It targets both package authors and users that would like to use this package manager to install software on their systems.

This package management system should run on any Linux machine. Due to the differences of the various Linux distributions (and versions of them) we cannot guarantee that all the packages will build without errors on every possible machine. We do however automatically test all packages on a set of test machines running CentOS. In the future additional test systems will be provided.

The DESDM package management system is based on EUPS 1.2.30. EUPS was not modified for DESDM, instead some additional scripts were written to support some of the use-cases.

Nomenclature

Term	Description
Product	A product is a name for a software component. Typical products are 'cfitsio', 'atlas', 'sextractor' and so on.
Package	A package is specific version of a product, compiled in a specific way, possibly linking to a specific set of libraries (other packages). Are are typically several packages for the same product.

Installing EUPS

Before any packages can be installed, EUPS must be installed. It its highly recommended to use our install script to do this, as it sets some additional environment variables which will be assumed to be present while working with the package manager. We do not support alternative installation methods.

The installation is a three-step procedure.

1. Make sure that the system has the prerequisites. This typically involves installing software that, for various reasons, are not managed as packages.
2. Download the install script.
3. Run the install script. The script will interactively ask for the installation path and such.

Each step is described in more detail below.

Prerequisites

If we would track every last bit of software in the package manager, we would effectively build our own Linux distribution. Instead we assume that a Linux distribution is already in place. This also allows everyone to use our package manager on their already set-up systems. Drawing a clear line between software maintained in our package system and software we assume to be present is not possible. Some guidelines are described in the package author section.

It turns out to be quite hard to provide an accurate list of prerequisites (for example: How to reliably detect the minimum version of libc needed by sextractor?). The following list is updated with the current state of our knowledge.

Dependency	Used by	RHEL / centOS package (if known)	OSX
python (some old version will do)	Package management system itself	python	-
wget	Package management system itself	wget	wget (install wget from source)
curl	Package management system itself	curl	curl (install instructions soon)
unzip	oracleclient 11.1.0.7	unzip	-
zlib	libpng 1.2.38, python 2.7.3	zlib zlib-devel	zlib
bzip2	python 2.7.3	bzip2 bzip2-devel	bzip2
libfreetype	plplot 5.8.0	freetype freetype-devel	freetype (install freetype from source)
libX11	plplot 5.9.9	libX11 libX11-devel	
libXau	plplot 5.9.9	libXau libXau-devel	
libxcb	plplot 5.9.9		
libXext	plplot 5.9.9	libXext libXext-devel	
libxpm	root 5.26.00	libXpm libXpm-devel	
libXft	root 5.26.00	libXft libXft-devel	
libcrypt	python 2.7.3	openssl openssl-devel glibc-devel	
libaio	oracleclient 11.1.0.7	libaio libaio-devel	
libfreebl3	python 2.7.3	nss-softokn-freebl nss-softokn-freebl-devel	
libnsl	python 2.7.3	glibc-devel	
gcc	various packages	gcc gcc-c++ make autoconf patch	Apple Xcode 5.1.1 (dmg)
gfortran	atlas 3.8.4, netlib-lapack 3.4.1	gcc-gfortran	version >= 4.8.2 (gfortran dmg)
pkg-config	glib 2.29.2	pkgconfig	install from source instructions
subversion	Package management system itself	svn (subversion on rhel5)	-
rsync	unitRunning 0.0.11	rsync	rsync
tar	desdm_eupsinstall.py	tar	

We optionally support Intel's ICC and MKL (see the next section)

Intel Compiler and Math Kernel Library

Some users are interested in using the compiler from Intel and Intel's math library. But since not everyone has access to those commercial products they are optional. If they are to be used, they need to be installed manually in advance. We cannot distribute ICC and MKL via EUPS for licensing reasons.

The following table lists the minimum versions required. Note that gcc and gfortran are still required since not all packages might support ICC and MKL (earlier versions of sextractor for example).

Dependency	Used by
icc (>=11.1)	sextractor (icc&mkl build only)
mkl (>=10.3)	sextractor (icc&mkl build only)

ICC and MKL can only be used in combination.

Downloading and running the installation script

The latest version of the installation script is available here: http://desbuild2.cosmology.illinois.edu/desdm_eupsinstall.py .

You can download it with wget (wget is a prerequisite, so you should have it installed by now).

```
$ wget http://desbuild2.cosmology.illinois.edu/desdm_eupsinstall.py
```

After that, run it with python:

```
$ python desdm_eupsinstall.py
```

and follow the instructions.

It will ask for two installation paths. One for EUPS itself and one for the software installed via EUPS packages. It will also ask if ICC and MKL should be used (which leads to a set of additional questions). The script can also modify the login-scripts to setup EUPS every time you log in. After confirming the settings the script will download and install EUPS.

During the installation procedure a bash and a c-shell script are generated which setup the environment variables required to use EUPS. The installation script output tells you where the files are and how to 'source' them.

The script performs some rather primitive checks on the prerequisite. There is no guarantee that all prerequisites are in place just because the script completes. It is just a help to detect some common issues.

Setup Subversion (SVN) access

The source code of some packages is directly fetched from subversion upon installation of the package. This requires read access to the subversion repository on https://dessvn.cosmology.illinois.edu/svn/desdm/devel/. Further more subversion must be configured in a way that allows access to this repository without interactively asking for the user's password. Otherwise the package installation will block for ever as EUPS does not support interaction with the user during a package installation.

The simplest way to set this up is to run the following command:

```
$ svn info $SVNROOT
```

If it just runs though and prints out some information about the current state of the repository then everything is setup correctly. If not, it will ask the user for input. Typically two questions appear:

- Error validating server certificate.... (R)eject, accept (t)emporarily or accept (p)ermanently? **Enter 'p' to ensure that the question won't reappear.**
- Username & Password: SVN asks first for the password, using the wrong username. Just press enter. It will then ask for the username and for the password. **Agree to store the password.**

Subversion (SVN) access problems

If you forgot your password, it might be stored as plain text in your home directory space under:

`$HOME/.subversion/auth/svn.simple/`

If you still have problems accessing SVN you can get help here filling out this form:

See if eups works

As a quick check to see if EUPS was properly installed and the environment is set-up correctly you can try

```
$ eups --version
EUPS Version: 1.2.30
```

also running the following command shows you all packages currently available:

```
$ eups distrib list
```

Check that the subversion repository can be accessed without interaction:

```
$ svn info $SVNROOT
```

This will print some information about the current state of the repository which isn't very interesting at this point. The idea is to check that this works without asking for any kind of input such as a password.

Installing Software with EUPS

Once EUPS is installed, installing packages should fairly easy. You tell EUPS the name of the product you'd like and which version and EUPS installs that piece of software. EUPS also recursively installs all dependencies in the proper order. After the product is installed it needs to be activated before it can be used. We call this 'setup'.

How to find the name and version of the software I'd like to install

The 'classical' way

Run this command:

```
$ eups distrib list
```

It lists all packages currently available in the package repository. While this works, it is often not that helpful to find the right package as there will be several versions of each product.

The DESDM way

As part of our continuous integration efforts web pages are generated automatically to reflect the current state of the package repository. They can be found here (might be worth a bookmark):

http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/products

On the left side is a list of all products. The matrix on the front page shows which versions of each product is used in a particular pipeline version. Clicking on a product name lists all the versions available of that product. Colors indicate if the automated build tests have found a problem with a particular package.

Installing by name and version

Once the name and version of the package is known, installing is just a single command:

```
$ eups distrib install NAME VERSION
```

for example to install version 2.17.0+0 of *sextractor* run

```
$ eups distrib install sextractor 2.17.0+0
```

It will automatically trigger the installation of atlas and fftw.

Installing by name and pipeline

There is also a little shortcut. For example if you would like to install the *sextractor* version used by firstcut-stable, you can directly run (pipeline names are always in upper-case):

```
$ eups distrib install sextractor -t FIRSTCUT-stable
```

EUPS will look up the version number for you, based on the -t argument. EUPS will behave exactly as if the package was installed by explicitly giving the version number.

Installing a complete pipeline

Instead of installing an individual package it is also possible to install all packages that belong to a certain pipeline with a single command:

```
$ eups distrib install FIRSTCUT -t FIRSTCUT-stable
```

This will make sure that all packages which are directly or indirectly used by firstcut-stable are installed.

Setting up by name and version

After installation the software is compiled and installed in its very own directory structure. Before it can be used the environment variables need to be set-up:

```
$ setup NAME VERSION
```

For sextractor this might look like this:

```
$ setup sextractor 2.17.0+0
```

This will ensure that the PATH environment variable includes the correct version of sextractor.

Setting up by name and pipeline

The same shortcut used for installing can also be used for setup:

```
$ setup sextractor -t FIRSTCUT-stable
```

A package can also be un-setuped. This helps to keep the environment variables clean.

```
$ unsetup sextractor
```

Setting up a complete pipeline

Corresponding to the install commands it is also possible to setup all packages that belong to a certain pipeline:

```
$ setup FIRSTCUT -t FIRSTCUT-stable
```

List all installed / set-up packages

To get a list of all packages currently installed on the local system execute:

```
$ eups list
```

Those that are currently setup are marked with 'setup'.

What to do if things go wrong?

Should a package fail to install properly, or if the environment gets wrongly setup for a package, please report it! One of the main motivations for this package management system is to solve each software deployment problem once in the consortium, instead of once per collaborator. If problems are reported, the package authors will look at it personally and make sure that the same problem will not occur anymore for anybody else.

Please report all issues to JIRA.

The JIRA is setup with several projects and components to which tickets can be added. The important thing is to have a ticket, we will move it should it land in the wrong component, as it is sometimes not clear where a problem fits.

We have a continuous integration system in place that automatically performs build-tests on a set of test machines. If you encounter problems building the packages on your machines it might make sense to dedicate a representative machine for continuous integration. That way the package authors get automatic feedback should a package fail on your machines.

The eups distrib install command hides the output of the build script, only printing the last few lines of the output in case of a failure. This makes it hard to track down the cause. The complete output is stored a log file in the directory:

```
$EUPS_PATH/EupsBuildDir/Linux64/[PRODUCT]/[VERSION]/[PRODUCT].log
```

This this directory is deleted if the install was successful, so the log file only exists on failure.

TODO: how to open a JIRA ticket for this

Some remarks for those with EUPS experience

The '-t' option used above stands for 'tag', a EUPS feature we use in a very specific way in DESDM. Special meta-products, typically named after pipelines, automatically create a tag for each package they, directly or indirectly, depend on. This allows the above use-cases where a pipeline name is used instead of a specific version number. Installing the meta-product itself installs the hole pipeline. In this case too it makes sense to use the '-t' trick to specify the version.

Creating new packages

The package generation procedure applied in DESDM is quite different from the one used on stock EUPS. For one thing you don't need any EUPS commands at all. The process is tightly integrated with our continuous build system.

Every package on our repository server is defined in subversion. The repository is server automatically synchronized with subversion. Thous to create a package in our repository, you need to make the right changes in subversion. You may use any subversion tools and methods you like to do this.

The package authors has to provide:

- The name of the product
- The version number of the package
- The build script
- The table file

The build script contains the instructions of how to build and install the package. The table file lists the dependencies and contains instructions to setup the environment variables for the software to work.

Versioning schema

The full identity of a package consists of the name and the version. The version is further divided in the product version and the package version. Optionally a configset name can be specified. The product version tracks changes in the code, where as the package version tracks changes in the build and table files. The configset is used if the same software is build in different variants (for example with different parameters passed to `./configure`).

The full identity of a package is given by:

- PRODUCTNAME PRODUCTVERSION+PACKAGEVERSION

in the usual case where no configset is required. If there is a configset it is appended to the product name with an underscore.

- PRODUCTNAME_CONFIGSET PRODUCTVERSION+PACKAGEVERSION

Lets look at a few examples:

	Product Name	Configset	Product Version	Package Version
cfitsio 3.300+0	cfitsio		3.300	0
libjpeg 6b+1	libjpeg		6b	1
altas_netlib 3.8.4+2	atlas	netlib	3.8.4	2

Product Name

The product name is an non-empty string that may contain the following characters:

- a-z, A-Z
- 0-9

No spaces, slashes, (under)scores, or any other special characters are permitted.

The package name should be all-lowercase. Camel-case can be used if required for readability.

Configset Name

Most packages won't have a configset name. The configset is used if the same software needs to be compiled in several variations. This is rarely needed. If possible one should try to create a single package that works for all the usecases.

configset names follow the name naming convention as the product name.

Product Version

A difference in the package version indicates a change in the source code.

The package version must be of the form `PrefixAAA.BBB.CCC-LLL.MMM.NNN`. All the components are optional. Each component may contain the following characters:

- a-z, A-Z
- 0-9

The product version `trunk` has a special meaning (see below) and should not be used for product releases.

The product version is typically selected by the developers of the software. For third-party libraries it is strongly recommended to use the version under which the software was released without any changes as the product version.

Package Version

If two packages only differ in the package version, the source code is the same, but either the build or the table files is different. The most common reasons to increase the package version are:

- Bug fixes in the build file or table file.
- Changes of the dependencies listed in the table file. Especially common is the case where one of the dependencies is replaced to a newer version of the same product.

At the moment the package version is a single integer which must increased at every change of the build or table file. Whenever the product version changes, the package version must be reset to zero.

Version comparison

Versions of the same product have to be compared in order to keep track of the latest one. Comparison is limited to versions of the same package. The sorting rules are as follows:

1. The elements are split on `.` (or `_, +, -`) and compared as integers or alphabets. In case if two elements are not same (for example '2a' and 'ab'), then the integer is given the preference over the alphabet otherwise each element(string on numbers) is compared with the other when same (in 'ab' and 'a2', 'ab' with 'a' and ''(nothing) with '2').
2. Alpha and beta versions are considered as earlier versions (3.14.0a < 3.14.0b < 3.14.0)
3. If the two product versions are equal then the packageversion decides the earlier version (3.14.0+1 < 3.14.0+2)

Build script

For DESDM we use EUPS in a rather simplistic way. Each package is build and installed by execution of a bash script written by the author. It contains the instructions to build the software from its source and install it. You can do about everything in that bash script. Typically the script will roughly perform the following steps:

1. Fetch the source code from somewhere
2. ./configure
3. make
4. make install

Take for example the build script of cftisio 3.300+0:

```
wget $EXTERNAL/$PRODUCT/$PRODUCT-$VERSION.tar.gz
tar xzf $PRODUCT-$VERSION.tar.gz
cd cfitsio
./configure --prefix=$PRODUCT_DIR
make
make install
```

The web-reports of the repository show the build script for every package http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/products . It might be a good idea to peek at similar packages to see how they are doing it.

There are some environment variables that can be assumed to be present in every build script:

Variable	Description	Example Value w/o configset	Example Value w. configset
EXTERNAL	URL to the place were the tarballs of external dependencies are stored.	http://desbuild2.cosmology.illinois.edu/eeups/webservice/resources/	http://desbuild2.cosmology.illinois.edu/eeups/webservice/resources/
SVNROOT	URL to the subversion repository.	https://dessvn.cosmology.illinois.edu/svn/desdm/devel/	https://dessvn.cosmology.illinois.edu/svn/desdm/devel/

SVN_PATH	Relative path to the root folder of the source inside the product's SVN directory.	tags/3.8.4+0	tags/3.8.4+0
PRODUCT	Name of the product. For historical reasons the configset is appended.	atlas	atlas_netlib
VERSION	Version of the product.	3.8.4	3.8.4
PKG_VERSION	'+' part of the package version.	1	1
FULL_VERSION	The complete package version	3.8.4+1	3.8.4+1
FLAVOR	The flavor of the system where the package is installed.	Linux64	Linux64
PRODUCT_DIR	The base directory into which the package must be installed.	[...]/Linux64/altas/3.8.4+1	[...]/Linux64/altas_netlib/3.8.4+1
[DEPENDENCY]_DIR (for example CFITSIO_DIR)	For every package this package depends on, there is at least this environment variable. It points to the directory that was the PRODUCT_DIR of that package.	[...]/Linux64/cfitsio/3.300+0	[...]/Linux64/cfitsio/3.300+0

If your package as dependencies (listed in the table file, see below), then all those packages will be installed and set-up before the script is executed. You can assume that all environment variables set by the table files of the dependencies are available.

You can also assume that all tools and libraries listed as prerequisites in the install chapter of this manual are present. You may also use tools that are available on every Linux distribution, such as tar.

Rules for build scripts

- You may use the current directory to store temporary files. For example to unpack and build the source. The directory will be deleted after the installation.
- Install into $PRODUCT_DIR. The directory is created before the build script is executed.
- Do NOT make ANY modifications outside either the current directory or the directory pointed to by $PRODUCT_DIR what so ever!.

Table file

The table files serves two purposes: It lists the dependencies of the package and it contains the instructions to setup the environment before the software in the package is used.

Note that even if the table file looks a bit like a bash script on first sight, they are not. Regular bash commands will not work.

Command	Description	Examples
setupRequired	Declares a dependency. Only direct dependencies need to be listed. EUPS will traverse them recursively.	setupRequired(libpng 1.2.38+0) setupRequired(libjpeg 6b+0)
envPrepend	Adds a path in front of an environment variable. If the variable is not yet set it is set to the given path.	envPrepend(PATH, ${PRODUCT_DIR}/bin)
envAppend	Adds a path to the end of an environment variable. If the variable is not yet set it is set to the given path.	endAppend(LD_LIBRARY_PATH, ${PRODUCT_DIR}/lib))

Rules for table files

- DO NOT USE version ranges. EUPS has some support to specify version ranges with setupRequirement. DESDM does not support this. Always specify the exact version. This ensures that we build packages exactly the same way everywhere, making installations reproducible.
- Always use curly brackets around variables. EUPS will not work without them.
- Use envPrepend instead of envAppend. This is important as it ensures that we really ending up using the software in the package and not some other installation that happens to be installed on the machine.

Testing a package locally

Before adding the package to subversion (and thereby to the package repository server) it is often advantageous to test the package first on the local machine. We provide a tool that creates and installs a package locally using the same methods used to create the packages in the repository.

The tool is available on the package repository itself. Thous, before it can be used it needs to be installed (once only) and set-up:

```
$ eups distrib install eeups -t EEUPS-stable
$ setup eeups -t EEUPS-stable
```

The tool is most simple to use if both the build and table file are in the same directory and are named PRODUCTNAME.build and PRODUCTNAME.table. This is typically the case when preparing a package following the subversion directory layout described below.

From the directory containing the build and table file run

```
$ eeups_build PRODUCTNAME
```

This creates, builds and installs the package locally. The version will be 'dev+0' to indicate that this package was not installed from the eups repository server.

The command has essentially the same effect as to commit the package to subversion, wait for it to appear on the package repository server and then execute 'eups distrib install PRODUCTNAME dev+0'. Only that neither subversion nor the package repository is changed in anyway (nothing is leaving the machine). Also it is not possible to add packages with 'dev' as their package version to the eups repository server.

The tool has some other options for example to install the package in a separate place instead of the default location. You may also specify other locations of the build and table files. Use

```
$ eeups_build --help
```

To get a documentation of all options.

The eeups_build command hides the output of the build script, only printing the last few lines of the output in case of a failure. This makes it hard to track down the cause. The complete output is stored a log file in the directory:

```
$EUPS_PATH/EupsBuildDir/Linux64/[PRODUCT]/[VERSION]/[PRODUCT].log
```

This this directory is deleted if the install was successful, so the log file only exists on failure.

Subversion directory layout

All packages in the package repository originate from subversion. Thous the proper way to modify the package repository is to modify the subversion repository. The package repository will be updated to reflect any changes in subversion. It is important to stick to this layout to ensure that the packages can be located by the synchronization scripts.

The required layout builds on top of the usual trunk/tags/branches layout commonly applied in subversion. The same layout is used for products where the source code is maintained in the subversion repository and for third-party products where the source code is stored outside. In the first case the code will be stored in parallel to the build & table file. In the second the build and table files will be the only files in the directory structure.

- For each product a directory must exist in subversion. It should, but must not, be named like the product (for example 'cfitsio'). The location of the directory is not important. It has to be stored in the configuration file (see below).
- Inside this product directory a sub-directory called 'tags' is expected.
- Inside 'tags' packages will be created for every sub-directory with a name that follows the PRODUCTVERSION+PACKAGEVERSION schema described in an earlier section.
- Within the tag the build and table files are expected. They have to be named as follows:
    - PRODUCTNAME.build and PRODUCTNAME.table for packages without a configset.
    - PRODUCTNAME_CONFIGSET.build and PRODUCTNAME_CONFIGSET.table for packages with a configset
- In the special case of trunk-packages (see below) the trunk directory is also scanned and a package will be created if a build and table file are present.

Example for cfitsio

```
.../cfitsio/
            tags/
                 3.280+0/
                         cfitsio.build
                         cfitsio.table
                 3.300+0/
                         cfitsio.build
                         cfitsio.table
                 3.300+1/
                         cfitsio.build
                         cfitsio.table
```

This would create three packages with versions 3.280+0, 3.300+0 and 3.300+1.

Example for atlas

```
.../atlas/
          tags/
               3.8.4+0/
                       atlas.build
                       atlas.table
                       atlas_netlib.build
                       atlas_netlib.table
               3.8.4+1/
                       atlas.build
                       atlas.table
                       atlas_netlib.build
                       atlas_netlib.table
```

This would create four packages, two without a configset and two with configset 'netlib'. The versions would be 3.8.4+0 and 3.8.4+1.

The layout is chosen such that one can simply keep the build and table file in the base directory of the source code (directly in trunk). They will end up in the right place once the trunk is tagged. One must follow the usual rule not to make changes into an existing tag. Instead make a new tag (typically with an incremented package version).

The location of this directory tree inside the subversion repository is not important. As a convention, all products where the build & table files are not maintained in the same directory tree as the source-code are located under https://dessvn.cosmology.illinois.edu/svn/desdm/devel/eupsScripts/external. Those are mostly third-party products and internal products where we'd like to keep code and build/table separated (at least for a while).

Internal and external Products

The above structure containing the build and table files can either be stored separtely from the source or it can be the same structure in which the source code is maintained. The later implies that the development team uses the desdm subversion repository to manage the source-code. Obviously this is not the case for all products which are not developed within the consortium, such as cfitsio. We call those external products. They still have their directory tree within the desdm repository, but it will only contain the build and table files.

If the build and table files are stored in the same directory tree as the source code, we call them internal products. Internal products have the advantage that the maintainance of the build and table file becomes a natural part of the software development. Developers are encuraged to keep an up-to-date version of the build and table file in the trunk directory, changing it along with the source code if necessary. This package management system will ignore those files within the trunk since it only looks for tags. The advantage of keeping them in the trunk directory is simple: Once the software is ready for another release, just create a tag in the usual subversion way, that is by copying the trunk into a tag directory. The package management system will detect that tag, and the build and table files within, and create a new package. This is quite handy for the develpers as they don't have to do any additional effort to create a package beyond creating the subversion tag that they have to create anyway.

There are a few special applications of the above. Some products are treated like external ones, even though their source code is managed in the same subversion repository. In other words such products have two directory trees: one storing the source code and one storing the build and table files. This creates a clear separation of source-code and package management. We apply this in cases where the development team does manage the build and table files them selves. As long as the new package management system is still in development, we do this for all packages. The goal is to migrate those into 'internal' products, with just one directory tree, as soon as the new system is ready and the development team were instructed.

Another, very rare, case are packages that are not developed in the consortium, but that have the source (or binaries, if the authors don't provide the source code) in our subversion repository anyway. The package maintainer needs to fetch the code from the authors and check it in. Typically the tarball it self is stored in subversion directly. The motivation is that the usual way, uploading the tarball to our webserver, has the draw back that the webserver is publically accessible, where as our subversion repository is protected. We only do this for products where the licence does not allow public redistribution.

Register a new product

Each product needs to be registered in the configuration file. It points the scripts to the directories where it will start scanning the directory layout described above for packages.

The configuration file is stored inside subversion it self: https://dessvn.cosmology.illinois.edu/svn/desdm/devel/eupsScripts/external/eups.cfg

In the [products] section, add a line for the new product. It must have three parts separated by one or more white-spaces:

1. Name of the product
2. Product directory in subversion. This is the path that points to the directory layout described above.
3. An email address of a person that can be contacted if there is a problem with the packages from this product.

There are many products already declared which can be used as a reference.

Package testing on dedicated test machines

In DESDM we apply continuous build. Each package on the package repository is automatically build on a set of test machines. Those test machines are selected to represent the various environments where the software needs to run. This gives the developers / package maintainers an early feedback should the package fail to build on a certain hardware type or linux distribution. We are currently building up the set of test machines used for this.

The results of the continuous build are displayed here:

http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/

The current version of the webreport has a small bug: Packages will always be marked OK (green) during the time after creation and before the build servers have completed their build-test (the package status is OK during that time because no error has been reported). Once the build results come in, the status might change to ERROR (red). Always check the details of the package to see if all builds have already completed.

Remember not to change an existing EUPS tag when fixing bugs reported in the build tests. Always create a new package. It is perfectly OK to have old packages fail in a product. Most products have, or will eventually have, such obsolete, failing packages. The webreports show the packages with the highest package versions (the number after the '+').

Trunk Packages

In general the package management system follows the principle that a package is never changed once created. Trunk-packages are the exception to this rule.

A trunk package is a package for the code that is currently in the `trunk` directory of that product's subversion tree. The package is named as follows:

- Product name: `imsupport`
- Product version: `trunk`
- Package version: `+0`

Since the trunk code is volatile, so is the package. The `+0` is never incremented!

Installing Trunk Packages

Trunk packages are installed the same way as all other packages:

```
eups distrib install imsupport trunk+0
```

The difference is that the current source code from the trunk svn-directory (instead of an svn-tag) is exported, built and installed (Note: the package author has to make this true in his build script). This implies that the time when the `eups distrib` command is executed matters. If a trunk package is installed on two machines on two different days, they will very likely build with different source code.

Updating Trunk Packages

With regular packages there is no update routine required, as they never change. One just installs the newer version of the product. Since trunk packages are changing, one might have the need to update it.

We are currently planning a tool for this. But for the moment the following procedure can be used:

```
eups undeclare imsupport trunk+0
eups distrib install imsupport trunk+0
```

This causes a re-install of the package (which will use the current code from subversion). Note that this procedure does not handle dependencies. One has to manually decide which depending packages need rebuilding as well. In the case of `imsupport` one might need to apply the same procedure to `imd` `etrend` afterwards as well.

Creating Trunk Packages

Trunk packages are equal to regular packages in almost all aspects. A `eups.cfg` configuration file must be placed in the product's subversion directory and the `trunk_package` flag must be set:

```
.../imsupport/
             eups.cfg
             tags/
                 ...
             trunk/
               imsupport.table
               imsupport.build
```

The content of the eups.cfg must contain:

```
[product]
trunk_package=True
```

Natrually the build script of a trunk package must export the source from the trunk. It might be helpful to use the `SVN_PATH` environment variable which is available in all build scripts.

```
svn export $SVNROOT/imsupport/$SVN_PATH imsupport
cd imsupport
./configure
make
make install
```

`$SVN_PATH` is set to `trunk` for trunk packages and set to `tags/$FULL_VERSION` for regular packages. This way the build script does not need to be changed when copying the trunk to a tag directory when releasing the software.

it is highly recommended to have the source code in the same trunk directory as the build and table files. Our continuous integration scripts can only detect a change to a package if the change happens in the directory where the build and table files are stored. This is called an 'Internal Product'. See above.

Depending on Trunk Packages

Using `setupRequired` packages can depend on each other. Trunk packages can depend on other packages as they wish, but there is one important rule:

Non-trunk package must NOT depend on trunk packages. A regular package represents a very specific, unchangable version of a software. This would no longer be given if somewhere in its dependency tree a package could change. Therefore they must never depend on trunk packages or we break provenance.

Trunk packages are allowed to depend on other trunk packages as well as regular packages.

Eventually the system will enforce this rule. For the moment this is in the responsibility of the package author.

Continuous build and test

If the source of a trunk package changes, that is, someone makes a commit to the trunk, then this triggers automatically a new build and a new run of the tests.

If other packages depend on this trunk package, then those are rebuild and retested as well. This results in true continuous integration testing, where the software is automatically tested not only for itself, but also if it still works together with the other packages.

Build and test reports are listed on the package's build information page in the Web-Reports.

Creating Meta-Products

Meta-Products simplify the selection of packages. On the package repository there is an ever increasing number of products available in an ever increasing number of versions. Some products might even be available in different configurations. This makes it difficult to find the right version, especially since several versions of a product might be in use at multiple places simultaneously (production, testing, development, ...).

A meta-product is a regular product with a special use-case and a few special conventions. Which will be explained in this section.

As with regular products, meta-products have packages. Those are called meta-packages. Meta-products typically don't contain any software (but they could). The main 'content' of a meta-package are its dependencies. The dependencies of a meta-package group together a set of other packages that form a logical unit. In DESDM we have a meta-product for each pipeline. It groups together (depends on) all the software required to run this pipeline.

Meta-Products serve two main use-cases:

- Installation of all packages required to run the complete pipeline.
- Installation of a single package out of a pipeline, by specifing the product name and the pipeline name. The exact version of the package is inferred from the pipeline.

Meta-package typically have an empty build script (but the file still needs to exist) and a comparatively large table file.

All indirect dependencies (packages on which packages in the meta-package's table file depend on) are automatically part of the meta-product.

Versioning Schema

Meta-packages follow the versioning schema of normal packages. However some additional guidelines exist:

Product Name

The name of a meta-product should be all upper-case to discriminate it from normal products which should be lower-case. Typically the name of the meta-product equals to the name of the pipeline (e.g. 'FIRSTCUT').

Configset Name

Configsets are currently not used for meta-products.

Product Version

The product version should represent the status of the pipeline. It could be a number, but also a descriptive word. For example:

- 'stable'
- 'production'
- 'beta'
- 'testing'
- 'development'

It probably makes sense to agree on a common set of product version descriptors for all pipelines. We still need to do this.

Package Version

Incremented integer as with normal packages.

As with regular packages, once a package is created it it never changed. With meta-products, changes typically result from 'promoting' a software package to a certain version of a pipeline. Iin other words, the dependencies of the meta-product change. As with regular packages, this results in an incremented package version.

Marking a product as 'meta'

Meta-products follow the exact same layout in subversion. To mark a product as a meta product add a 'eups.cfg' file into the product's root directory in subversion:

```
.../FIRSTCUT/
             eups.cfg
             tags/
                 stable+0/
                         cfitsio.build
                         cfitsio.table
                 stable+1/
                         cfitsio.build
                         cfitsio.table
                 development+100/
                         cfitsio.build
                         cfitsio.table
```

The content of the eups.cfg must be:

```
[product]
meta_product=True
```

Some remarks for those with EUPS experience

The DESDM system automatically creates a EUPS-tag for each meta-product and product version.For example there might be a tag for 'FIRSTCUT-stable'.

All packages, directly or indirectly, referenced from the meta-package with the highest package version is added to that EUPS-tag. For example the tag 'FIRSTCUT-stable' might correspond to the package 'FIRSTCUT table+5', if there is no other package with a higher package version.

The EUPS-tags are the only component in the package management system which changes over time (they are replaced when a package with a higher package version is created). This is ultimatively the reason why a user can write for example

```
eups distrib install FIRSTCUT -t FIRSTCUT-stable
```

and be sure to get the 'current' versions without explictly knowing them.

Package Life Cycle Qualifiers

Each package is given one of the following 'life cycle' qualifiers:

- OPERATIONAL (O)
- DEPRECATED (D)
- EXPERIMENTAL (E)

Based on these qualifiers, the the continuous integration build system can perform validation checks, can set tags for eups packages distributed through the distrib server. Furthermore, the webpages can be filtered for specific qualifiers - hence, help navigating in the thousands of packages we now have in the system.

Declaration of Life Cycle Qualifiers

Default qualifiers:

- Packages defined in the tags directory (of a given product) are identified as OPERATIONAL.
- Packages defined in the trunk or branches directory (of the given product) are identified as EXPERIMENTAL.

These defaults can be overruled for packages defined in the tags directory as follows:

- DEPRECATED packages can explicitly be declared in the eups.cfg file defined at product level.
- EXPERIMENTAL packages can be declared by appending an 'E' to the package version (e.g. eeups-1.4.3+4E). Note that this is not necessary for trunk and branch packages.

DEPRECATED packages can be declared in the eups.cfg file by the following syntax:

```
deprecated = 1.2.3+10; 3.2.1+0; [5.2+3,6.3+2]; [6.5+0,6.6+3); [8.8+1,*), (*,1.2+1]
```

Actually, for the ranges, a suitable ordering of the versions is adopted (the same ordering that you can see on the per product view in the eeups dashboard. Angular brackets indicate that boundary versions included, round brackets that boundary versions are excluded.

The * in [8.8+1,*) means that any version above (and including) 8.8+1 is included or, accordingly in (*,1.2+1] that any version below (and including) 1.2+1 is included.

Different versions or ranges of versions specified per qualifier are separated by a semi-colon (;).

Validation Checks

The validation checks will be based on the following rules:

- a package with qualifier O must not depend on E. Otherwise, the package will receive status=ERROR
- a package with qualifier O should not depend on D. Otherwise, the package will receive status=WARNING.

The result of these checks will be reflected in the package state and the package states are presented in the EEUPS Dashboard web pages.

Life Cycle Qualifiers as EUPS Tags

Furthermore, the qualifiers are reflected on the EUPS Distribution Server in form of according EUPS tags. It means that by eups distrib list -t 'OPERATIONAL' the operational versions can be listed. A dedicated package distribution service should serve exclusively only OPERATIONAL package versions to the DES user community.

Not yet implemented.

EEUPS Dashboard Views

The EEUPS Dashboard (see http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/) views can be filtered for these qualifiers (OPERATIONAL, EXPERIMENTAL, DEPRECATED, ALL). By default, the filter will be set to OPERATIONAL.

Not yet implemented.

EUPS User Utilities

User utilities support users in interacting with the EUPS system. Typical examples are utilities for creating new packages or for propagating version updates of a give package all the way up through the dependency tree.

All these tools are shipped as part of the eeups product:

```
$ eups distrib install eeups -t EEUPS-stable
$ setup eeups -t EEUPS-stable
```

or for a specific version to obtain new features in beta release status:

```
$ eups distrib install eeups 1.4.3+2
$ setup eeups 1.4.3+2
```

With the exception of trunk packages (see below), packages never change. If one releaes a new version of product, then all the other products that depend on it likely need a new + version as well, as one wishes to update its dependency. If a package 'at the bottom' of the dependency graph is released, many such new packages may have to be created. This is a time consuming and error-prone process to do manually.

Two different versions of package propagation tools exist:

- Dependency Updater (pkg_propagator): Used for updating updating a given package for version updates of a direct or indirect dependency.
- Bulk Propagator (eeups_propagator): Used for propagating version updates in the bulk of all packages.

The following rules hold true for both propagator tools:

- Existing packages are not modified.
- For each product-productversion at most one new + version is introduced, even if several changes have to be applied to it.
- Newly introduced + versions are copies of the previous + versions. The only changes are applied to the table files (`setupRequired(..)` entries) to reflect changes in the dependencies.
- No new dependencies are added, nor are dependencies removed. They may be replaced by different versions.

Dependency Updater

Running The Dependency Updater

The typical user scenario looks as follows: Create a new version of a given 'reference' package by replacing one or several of its direct or indirect dependencies by a new version.

Let's give a hypothetical example. Assume that you would like to to create a new diffimg version by replacing in diffimg-8.0.28+2 the indirect dependency python-2.7.6+1 by the updated version python-2.7.7+0. This will assume that you have already added python-2.7.7+0 into the system (which is not at the time of writing). Then you can run the following command:

```
$ pkg_propagator --replace python-2.7.6+1 --by python-2.7.7+0 --reference diffimg-8.0.28+2
```

This results in the following steps:

Step 1: Identify new package versions

The tool does not interact with a local eups repository. Hence, you don't need to first have the 'old' diffimg version installed in your environment. Rather, the tool fetches the information on new packages to be created from the webserver () and prints them to the console.

In the given example, the relevant parts in the dependency graph are

```
diffimg-8.0.28+2
--- coreUtils-0.5.2+4
     --- psycopg-2.4.6+4
          --- python-2.7.6+1
     --- cxOracle-5.1.2+8
          --- python-2.7.6+1
```

The new packages proposed by the system would be the following:

```
diffimg-8.0.28+3E
--- coreUtils-0.5.2+5E
     --- psycopg2-2.4.6+5E
     --- cxOracle-5.1.2+9E
```

where psycopg-2.4.6+5E and cxOracle-5.1.2+9E would have dependencies on python-2.7.7+0. The 'E' appended to the package version indicates that by default EXPERIMENTAL packages are created. On the console, you would see something like:

```
cxOracle-5.1.2+8             ==> cxOracle-5.1.2+9E
psycopg2-2.4.6+4            ==> psycopg2-2.4.6+5E
coreUtils-0.5.2+4             ==> coreUtils-0.5.2+5E
diffimg-8.0.28+2              ==> diffimg-8.0.28+3E
```

Note that if there would already be a psyopg2-2.4.6 version with the same dependencies as psycopg-2.4.6+4 but the new python version and identical build file there would be no need to create a new package version for psyopg2-2.4.6. This is recognized by the system and no new psyopg2-2.4.6 package version would be proposed. This means that if you specify a reference and a package to be replaced for which a propagated version already exists no propagation is performed.

Furthermore, it gives you a warning for those 'existing' packages (in the propagated graph) that have not status OK (or INCOMPLETE). Propagating packages that have already some known problem in one of its dependencies might not be a good thing to do.

Step 2: Checkout old packages and create the new packages

On the command line you are asked whether you want to proceed with the checkout. When confirming that with 'yes' the old package versions to be replaced are checked out from SVN to the current directory - unless another directory has been specified with the --workdir parameter (see below). Then, the proposed new package versions are created in the local checkout directory. Note that changes are **not** committed automatically!

Step 3: Manually check/modifiy the changes

After the checkout has completed and the changes have been applied to the working copies, the `pkg_propagator` tool exits. It does not commit the changes directly.

The user can now look at the modified working copies and inspect the work done by `pkg_propagator`. All changes are already marked with `svn add` and are ready for commit.

It is perfectly possible to make manual changes at this point. The directories are regular subversion working copies. This can be very handy, for example to manually fix the problems reported in step 1.

Step 4: Commit new packages

Commit changes by executing the commit bash script:

```
$ ./commit "New diffimg 8.0.28 package version with python updated to 2.7.7."
```

Further Options

Further command line options available for the pkg_propagator:

Parameter	
--workdir	Allows you to specify a working directory where the old package versions are checked out from SVN and new package versions will be created. Defaults to the current directory.
--operational	Once set new 'operational' package versions are proposed (without 'E' appended to the package version).

--webservice	URL to the webservice from where the information on new package versions can be obtained. Defaults to http://desbuild2.cosmology.illinois.edu/eeups/webservice/propagator . Note that this service can be called directly from a web browser. By entering a URL of the form
	
	`<webservice>/E/<replace>/<by>/<reference>`
	
	or
	
	`<webservice>/O/<replace>/<by>/<reference>`
	
	for 'experimental' or 'operational' package versions, respectively. In the angular brackets <replace>, <by>, <reference> you need to specify package versions such as python-2.7.6+1. As a result, you will see a json file with all the old ('orig') packages and the corresponding new ('prop') packages to be created.
	
	Example:
	
	`http://desbuild2.cosmology.illinois.edu/eeups/webservice/propagator/E/atlas_netlib-3.8.4+6/atlas_netlib-3.8.4+7/eeups-1.4.3+2`

As usual, use

```
$ pkg_propagator --help
```

to get help on the command line.

User Scenarios

Create New Experimental Package Version (with dependency update)

Follow the steps described above without setting the --operational parameter. With the above example:

```
$ pkg_propagator --replace python-2.7.6+1 --by python-2.7.7+0 --reference diffimg-8.0.28+2
```

This would, as described above, create the packages diffimg-8.0.28+3E, coreUtils-0.5.2+5E, psycopg2-2.4.6+5E, cxOracle-5.1.2+9E.

The package diffimg-8.0.28+3E would then be the package to test and play around with.

Create Operational Package Version associated with the Experimental Package Version

Possibly, once all the tests have successfully been concluded, you would like to create an operational version from that. Currently, the suggested way to do that is to again run the package propagator by using exactly the same parameters as above - but by setting an additional --operational flag:

```
$ pkg_propagator --replace python-2.7.6+1 --by python-2.7.7+0 --reference diffimg-8.0.28+2 --operational
```

This would propose to create new packages diffimg-8.0.28+4, coreUtils-0.5.2+6, psycopg2-2.4.6+6, cxOracle-5.1.2+10 - as long as no other package versions have been added for the given product versions (diffimg-8.0.28, coreUtils-0.5.2, psycopg2-2.4.6, cxOracle-5.1.2) in the meantime.

In the future, we may simplify that - e.g. by a command of the form

`pkg_propagator diffimg-8.0.28+3E --operational`

which would actually create new package versions diffimg-8.0.28+3, coreUtils-0.5.2+5, psycopg2-2.4.6+5, cxOracle-5.1.2+9 (obtained from the experimental versions by removing the 'E'). in any case, the experimental version will remain in the system.

Known Issues

- Currently, only one single dependency can be replaced at the time. Work in progress to replace multiple packages at the same time.

- No transaction handling: The propagator interacts with both, the continuous integration build system (for fetching the information which packages to create) and SVN for the checkout and the commit. The continuous build system needs some time to update its state for SVN changes (at the order of 30-60 secs) so that SVN and continuous integration build system are not guaranteed to always by concurrent. As a result, users may experience problems when submitting proposed changes - e.g. when two users want to do the same thing at the same time.

Bulk Propagator

Since the tool can easily create hundreds of packages, it is important to understand how the tool works before it is applied.

The tool operates in several steps:

Step 1: Tell eeups_propagate what to do

First the eeups_propagator analyzes the dependency graph and the arguments given to it, and decides which new packages should be created and what changes should be made their dependencies.

The following always holds:

- No existing packages are changed in any way.
- For each product-productversion at most one new + version is introduced, even if several changes have to be applied to it.
- Newly introduced + versions are a copy of the previous + version.
- The only changes made are to the `setupRequired(..)` entries in the table file.
- No new dependencies are added, nor are dependencies removed. They may be replaced by different versions.

Step 1.1: Most-recent dependencies only

The `eeups_propagator` updates the dependencies of every most-recent + version that has a dependency to a not-most-recent + version. This happens if no options are given to `eeups_propagator`:

```
$ eeups_propagator
```

For example, say the following packages exist:

```
python-2.7.3+0
python-2.7.3+1
python-2.7.3+2
pyfits-3.0.7+0   depends on python 2.7.3+0
pyfits-3.0.8+0   depends on python 2.7.3+0
pyfits-3.0.8+1   depends on python 2.7.3+1
```

Then two new packages would have to be created:

```
pyfits-3.0.7+1   depends on python-2.7.3+2
pyfits-3.0.8+2   depends on python-2.7.3+2
```

`eeups_propagator` *always* performs those changes as there is little point in having most-recent packages depend on packages with out-dated build and table files.

Such changes propagate through the dependency graph. If there is a package that depends on `pyfits-3.0.8+1` a new version of it would be created as well, pointing to `pyfits-3.0.8+2` and so on.

Step 1.2: Replace a product version

Using the `--replace oldpackage newpackage` option, `eeups_propagator` can also be used if a newly released package has not just changed its package version, but also the product version (the part before the +).

For example, say a problem with perl 5.10.1 forces us to switch to perl 5.18.1. A package `perl-5.18.1+0` is created that should replace the old `perl-5.10.1+1` which is currently used by countless other packages. By default `eeups_propagator` will not do anything here if `perl-5.10.1+1` is the most recent + version of perl.5.10.1. In this scenario we can apply the `--replace` option:

```
$ eeups_propagator --replace perl-5.10.1+1 perl-5.18.1+0
```

Now `eeups_propagator` will 'replace' all dependencies to `perl-5.10.1+1` with dependencies to `perl-5.18.1+0`, meaning that it will create a new + version for any package that depends on the old perl package.

Of course the change is propagated upwards through the dependency graph.

Step 2: Plan the changes and verify consistency

Once `eeups_propagator` is invoked, it uses the information given to it to assemble a list of all the packages to be created. It runs the same verification routines on them (locally), that would be run on the server if the packages would be created on the eups repository at this stage. This detects if the changes introduce cyclic dependencies or version conflicts of any kind.

If a problem is encountered, the list of all planned changes is printed out, together with a list of the problems found, and the user is asked if he would like to continue anyway.

if the `--askfirst` option is given, the user is asked for permission to continue even if no problems were found.

Step 3: Checking out and applying changes

`eeups_propagator` will now internally run a `svn checkout` for each affected product. The working copies are created in the local directory, unless specified differently with `--work`. It checks out the URL of the product specified in the `eups.cfg` file, that is, it checks out the directory tree under which the build and table files are stored.

The script then applies the changes as planned. It internally uses `svn copy` to create new packages and then modifies the table file as required.

The changes are **not** commited automatically!

Step 4: Manually check/modifiy the changes

After the checkout has completed and the changes have been applied to the working copies, the `eeups_propagator` tool exits. It does not commit the changes directly.

The user can now look at the modified working copies and inspect the work done by `eeups_propagator`. All changes are already marked with `svn add` and are ready for commit.

It is perfectly possible to make manual changes at this point. The directories are regular subversion working copies. This can be very handy, for example to manually fix the problems reported in step 2.

Step 5: Commit

Once sufficiently convinced that the changes are fine, they can be commited.

Since there might be a rather large number of working copies that need submitting, eeups_propagator has already prepared a bash script for this.

The script is called `commit` and is located in the same directory as the checked out working copies. Invoke it with the commit message as argument:

```
$ ./commit "Replaced perl-5.10.1+1 with perl-5.18.1+0 in all most recent \+ versions."
```

The new packages should appear in the eups repository and web pages after the usual delays.

Remember: If something goes very wrong, the changes can always be undone thanks to subversions versioning. See Red Book (scroll down to "Undoing Changes").

User's Guide

For an abridged version (but not a replacement) of this guide see The Impatient's Guide to DESDM EUPS installation, which contains specific information about installing on OSX (10.7-10.11).

This is the user manual for DESDM's new package management system. It targets both package authors and users that would like to use this package manager to install software on their systems.

This package management system should run on any Linux machine. Due to the differences of the various Linux distributions (and versions of them) we cannot guarantee that all the packages will build without errors on every possible machine. We do however automatically test all packages on a set of test machines running CentOS. In the future additional test systems will be provided.

The DESDM package management system is based on EUPS 1.2.30. EUPS was not modified for DESDM, instead some additional scripts were written to support some of the use-cases.

Nomenclature

Term	Description
Product	A product is a name for a software component. Typical products are 'cfitsio', 'atlas', 'sextractor' and so on.
Package	A package is specific version of a product, compiled in a specific way, possibly linking to a specific set of libraries (other packages). Are are typically several packages for the same product.

Installing EUPS

Before any packages can be installed, EUPS must be installed. It its highly recommended to use our install script to do this, as it sets some additional environment variables which will be assumed to be present while working with the package manager. We do not support alternative installation methods.

The installation is a three-step procedure.

1. Make sure that the system has the prerequisites. This typically involves installing software that, for various reasons, are not managed as packages.
2. Download the install script.
3. Run the install script. The script will interactively ask for the installation path and such.

Each step is described in more detail below.

Prerequisites

If we would track every last bit of software in the package manager, we would effectively build our own Linux distribution. Instead we assume that a Linux distribution is already in place. This also allows everyone to use our package manager on their already set-up systems. Drawing a clear line between software maintained in our package system and software we assume to be present is not possible. Some guidelines are described in the package author section.

It turns out to be quite hard to provide an accurate list of prerequisites (for example: How to reliably detect the minimum version of libc needed by sextractor?). The following list is updated with the current state of our knowledge.

Dependency	Used by	RHEL / centOS package (if known)	OSX
python (some old version will do)	Package management system itself	python	-

wget	Package management system itself	wget	wget (install wget from source)
curl	Package management system itself	curl	curl (install instructions soon)
unzip	oracleclient 11.1.0.7	unzip	-
zlib	libpng 1.2.38, python 2.7.3	zlib zlib-devel	zlib
bzip2	python 2.7.3	bzip2 bzip2-devel	bzip2
libfreetype	plplot 5.8.0	freetype freetype-devel	freetype (install freetype from source)
libX11	plplot 5.9.9	libX11 libX11-devel	
libXau	plplot 5.9.9	libXau libXau-devel	
libxcb	plplot 5.9.9		
libXext	plplot 5.9.9	libXext libXext-devel	
libxpm	root 5.26.00	libXpm libXpm-devel	
libXft	root 5.26.00	libXft libXft-devel	
libcrypt	python 2.7.3	openssl openssl-devel glibc-devel	
libaio	oracleclient 11.1.0.7	libaio libaio-devel	
libfreebl3	python 2.7.3	nss-softokn-freebl nss-softokn-freebl-devel	
libnsl	python 2.7.3	glibc-devel	
gcc	various packages	gcc gcc-c++ make autoconf patch	Apple Xcode 5.1.1 (dmg)
gfortran	atlas 3.8.4, netlib-lapack 3.4.1	gcc-gfortran	version >= 4.8.2 (gfortran dmg)
pkg-config	glib 2.29.2	pkgconfig	install from source instructions
subversion	Package management system itself	svn (subversion on rhel5)	-
rsync	unitRunning 0.0.11	rsync	rsync
tar	desdm_eupsinstall.py	tar	

We optionally support Intel's ICC and MKL (see the next section)

Intel Compiler and Math Kernel Library

Some users are interested in using the compiler from Intel and Intel's math library. But since not everyone has access to those commercial products they are optional. If they are to be used, they need to be installed manually in advance. We cannot distribute ICC and MKL via EUPS for licensing reasons.

The following table lists the minimum versions required. Note that gcc and gfortran are still required since not all packages might support ICC and MKL (earlier versions of sextractor for example).

Dependency	Used by
icc (>=11.1)	sextractor (icc&mkl build only)
mkl (>=10.3)	sextractor (icc&mkl build only)

ICC and MKL can only be used in combination.

Downloading and running the installation script

The latest version of the installation script is available here: http://desbuild2.cosmology.illinois.edu/desdm_eupsinstall.py .

You can download it with wget (wget is a prerequisite, so you should have it installed by now).

```
$ wget http://desbuild2.cosmology.illinois.edu/desdm_eupsinstall.py
```

After that, run it with python:

```
$ python desdm_eupsinstall.py
```

and follow the instructions.

It will ask for two installation paths. One for EUPS itself and one for the software installed via EUPS packages. It will also ask if ICC and MKL should be used (which leads to a set of additional questions). The script can also modify the login-scripts to setup EUPS every time you log in. After confirming the settings the script will download and install EUPS.

During the installation procedure a bash and a c-shell script are generated which setup the environment variables required to use EUPS. The installation script output tells you where the files are and how to 'source' them.

The script performs some rather primitive checks on the prerequisite. There is no guarantee that all prerequisites are in place just because the script completes. It is just a help to detect some common issues.

Setup Subversion (SVN) access

The source code of some packages is directly fetched from subversion upon installation of the package. This requires read access to the subversion repository on https://dessvn.cosmology.illinois.edu/svn/desdm/devel/. Further more subversion must be configured in a way that allows access to this repository without interactively asking for the user's password. Otherwise the package installation will block for ever as EUPS does not support interaction with the user during a package installation.

The simplest way to set this up is to run the following command:

```
$ svn info $SVNROOT
```

If it just runs though and prints out some information about the current state of the repository then everything is setup correctly. If not, it will ask the user for input. Typically two questions appear:

- Error validating server certificate.... (R)eject, accept (t)emporarily or accept (p)ermanently? **Enter 'p' to ensure that the question won't reappear.**
- Username & Password: SVN asks first for the password, using the wrong username. Just press enter. It will then ask for the username and for the password. **Agree to store the password.**

Subversion (SVN) access problems

If you forgot your password, it might be stored as plain text in your home directory space under:

`$HOME/.subversion/auth/svn.simple/`

If you still have problems accessing SVN you can get help here filling out this form:

See if eups works

As a quick check to see if EUPS was properly installed and the environment is set-up correctly you can try

```
$ eups --version
EUPS Version: 1.2.30
```

also running the following command shows you all packages currently available:

```
$ eups distrib list
```

Check that the subversion repository can be accessed without interaction:

```
$ svn info $SVNROOT
```

This will print some information about the current state of the repository which isn't very interesting at this point. The idea is to check that this works without asking for any kind of input such as a password.

Installing Software with EUPS

Once EUPS is installed, installing packages should fairly easy. You tell EUPS the name of the product you'd like and which version and EUPS installs that piece of software. EUPS also recursively installs all dependencies in the proper order. After the product is installed it needs to be activated before it can be used. We call this 'setup'.

How to find the name and version of the software I'd like to install

The 'classical' way

Run this command:

```
$ eups distrib list
```

It lists all packages currently available in the package repository. While this works, it is often not that helpful to find the right package as there will be several versions of each product.

The DESDM way

As part of our continuous integration efforts web pages are generated automatically to reflect the current state of the package repository. They can be found here (might be worth a bookmark):

http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/products

On the left side is a list of all products. The matrix on the front page shows which versions of each product is used in a particular pipeline version. Clicking on a product name lists all the versions available of that product. Colors indicate if the automated build tests have found a problem with a particular package.

Installing by name and version

Once the name and version of the package is known, installing is just a single command:

```
$ eups distrib install NAME VERSION
```

for example to install version 2.17.0+0 of *sextractor* run

```
$ eups distrib install sextractor 2.17.0+0
```

It will automatically trigger the installation of atlas and fftw.

Installing by name and pipeline

There is also a little shortcut. For example if you would like to install the *sextractor* version used by firstcut-stable, you can directly run (pipeline names are always in upper-case):

```
$ eups distrib install sextractor -t FIRSTCUT-stable
```

EUPS will look up the version number for you, based on the -t argument. EUPS will behave exactly as if the package was installed by explicitly giving the version number.

Installing a complete pipeline

Instead of installing an individual package it is also possible to install all packages that belong to a certain pipeline with a single command:

```
$ eups distrib install FIRSTCUT -t FIRSTCUT-stable
```

This will make sure that all packages which are directly or indirectly used by firstcut-stable are installed.

Setting up by name and version

After installation the software is compiled and installed in its very own directory structure. Before it can be used the environment variables need to be set-up:

```
$ setup NAME VERSION
```

For sextractor this might look like this:

```
$ setup sextractor 2.17.0+0
```

This will ensure that the PATH environment variable includes the correct version of sextractor.

Setting up by name and pipeline

The same shortcut used for installing can also be used for setup:

```
$ setup sextractor -t FIRSTCUT-stable
```

A package can also be un-setuped. This helps to keep the environment variables clean.

```
$ unsetup sextractor
```

Setting up a complete pipeline

Corresponding to the install commands it is also possible to setup all packages that belong to a certain pipeline:

```
$ setup FIRSTCUT -t FIRSTCUT-stable
```

List all installed / set-up packages

To get a list of all packages currently installed on the local system execute:

```
$ eups list
```

Those that are currently setup are marked with 'setup'.

What to do if things go wrong?

Should a package fail to install properly, or if the environment gets wrongly setup for a package, please report it! One of the main motivations for this package management system is to solve each software deployment problem once in the consortium, instead of once per collaborator. If problems are reported, the package authors will look at it personally and make sure that the same problem will not occur anymore for anybody else.

Please report all issues to JIRA.

The JIRA is setup with several projects and components to which tickets can be added. The important thing is to have a ticket, we will move it should it land in the wrong component, as it is sometimes not clear where a problem fits.

We have a continuous integration system in place that automatically performs build-tests on a set of test machines. If you encounter problems building the packages on your machines it might make sense to dedicate a representative machine for continuous integration. That way the package authors get automatic feedback should a package fail on your machines.

The eups distrib install command hides the output of the build script, only printing the last few lines of the output in case of a failure. This makes it hard to track down the cause. The complete output is stored a log file in the directory:

```
$EUPS_PATH/EupsBuildDir/Linux64/[PRODUCT]/[VERSION]/[PRODUCT].log
```

This this directory is deleted if the install was successful, so the log file only exists on failure.

TODO: how to open a JIRA ticket for this

Some remarks for those with EUPS experience

The '-t' option used above stands for 'tag', a EUPS feature we use in a very specific way in DESDM. Special meta-products, typically named after pipelines, automatically create a tag for each package they, directly or indirectly, depend on. This allows the above use-cases where a pipeline name is used instead of a specific version number. Installing the meta-product itself installs the hole pipeline. In this case too it makes sense to use the '-t' trick to specify the version.

Creating new packages

The package generation procedure applied in DESDM is quite different from the one used on stock EUPS. For one thing you don't need any EUPS commands at all. The process is tightly integrated with our continuous build system.

Every package on our repository server is defined in subversion. The repository is server automatically synchronized with subversion. Thous to create a package in our repository, you need to make the right changes in subversion. You may use any subversion tools and methods you like to do this.

The package authors has to provide:

- The name of the product
- The version number of the package
- The build script
- The table file

The build script contains the instructions of how to build and install the package. The table file lists the dependencies and contains instructions to setup the environment variables for the software to work.

Versioning schema

The full identity of a package consists of the name and the version. The version is further divided in the product version and the package version. Optionally a configset name can be specified. The product version tracks changes in the code, where as the package version tracks changes in the build and table files. The configset is used if the same software is build in different variants (for example with different parameters passed to `./configure`).

The full identity of a package is given by:

- PRODUCTNAME PRODUCTVERSION+PACKAGEVERSION

in the usual case where no configset is required. If there is a configset it is appended to the product name with an underscore.

- PRODUCTNAME_CONFIGSET PRODUCTVERSION+PACKAGEVERSION

Lets look at a few examples:

	Product Name	Configset	Product Version	Package Version
cfitsio 3.300+0	cfitsio		3.300	0
libjpeg 6b+1	libjpeg		6b	1
altas_netlib 3.8.4+2	atlas	netlib	3.8.4	2

Product Name

The product name is an non-empty string that may contain the following characters:

- a-z, A-Z
- 0-9

No spaces, slashes, (under)scores, or any other special characters are permitted.

The package name should be all-lowercase. Camel-case can be used if required for readability.

Configset Name

Most packages won't have a configset name. The configset is used if the same software needs to be compiled in several variations. This is rarely needed. If possible one should try to create a single package that works for all the usecases.

configset names follow the name naming convention as the product name.

Product Version

A difference in the package version indicates a change in the source code.

The package version must be of the form `PrefixAAA.BBB.CCC-LLL.MMM.NNN`. All the components are optional. Each component may contain the following characters:

- a-z, A-Z
- 0-9

The product version `trunk` has a special meaning (see below) and should not be used for product releases.

The product version is typically selected by the developers of the software. For third-party libraries it is strongly recommended to use the version under which the software was released without any changes as the product version.

Package Version

If two packages only differ in the package version, the source code is the same, but either the build or the table files is different. The most common reasons to increase the package version are:

- Bug fixes in the build file or table file.
- Changes of the dependencies listed in the table file. Especially common is the case where one of the dependencies is replaced to a newer version of the same product.

At the moment the package version is a single integer which must increased at every change of the build or table file. Whenever the product version changes, the package version must be reset to zero.

Version comparison

Versions of the same product have to be compared in order to keep track of the latest one. Comparison is limited to versions of the same package. The sorting rules are as follows:

1. The elements are split on `.` (or `_, +, -`) and compared as integers or alphabets. In case if two elements are not same (for example '2a' and 'ab'), then the integer is given the preference over the alphabet otherwise each element(string on numbers) is compared with the other when same (in 'ab' and 'a2', 'ab' with 'a' and ''(nothing) with '2').
2. Alpha and beta versions are considered as earlier versions (3.14.0a < 3.14.0b < 3.14.0)
3. If the two product versions are equal then the packageversion decides the earlier version (3.14.0+1 < 3.14.0+2)

Build script

For DESDM we use EUPS in a rather simplistic way. Each package is build and installed by execution of a bash script written by the author. It contains the instructions to build the software from its source and install it. You can do about everything in that bash script. Typically the script will roughly perform the following steps:

1. Fetch the source code from somewhere
2. ./configure
3. make
4. make install

Take for example the build script of cftisio 3.300+0:

```
wget $EXTERNAL/$PRODUCT/$PRODUCT-$VERSION.tar.gz
tar xzf $PRODUCT-$VERSION.tar.gz
cd cfitsio
./configure --prefix=$PRODUCT_DIR
make
make install
```

The web-reports of the repository show the build script for every package http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/products . It might be a good idea to peek at similar packages to see how they are doing it.

There are some environment variables that can be assumed to be present in every build script:

Variable	Description	Example Value w/o configset	Example Value w. configset
EXTERNAL	URL to the place were the tarballs of external dependencies are stored.	http://desbuild2.cosmology.illinois.edu/eeups/webservice/resources/	http://desbuild2.cosmology.illinois.edu/eeups/webservice/resources/
SVNROOT	URL to the subversion repository.	https://dessvn.cosmology.illinois.edu/svn/desdm/devel/	https://dessvn.cosmology.illinois.edu/svn/desdm/devel/
SVN_PATH	Relative path to the root folder of the source inside the product's SVN directory.	tags/3.8.4+0	tags/3.8.4+0
PRODUCT	Name of the product. For historical reasons the configset is appended.	atlas	atlas_netlib
VERSION	Version of the product.	3.8.4	3.8.4
PKG_VERSION	'+' part of the package version.	1	1
FULL_VERSION	The complete package version	3.8.4+1	3.8.4+1
FLAVOR	The flavor of the system where the package is installed.	Linux64	Linux64
PRODUCT_DIR	The base directory into which the package must be installed.	[...]/Linux64/altas/3.8.4+1	[...]/Linux64/altas_netlib/3.8.4+1
[DEPENDENCY]_DIR (for example CFITSIO_DIR)	For every package this package depends on, there is at least this environment variable. It points to the directory that was the PRODUCT_DIR of that package.	[...]/Linux64/cfitsio/3.300+0	[...]/Linux64/cfitsio/3.300+0

If your package as dependencies (listed in the table file, see below), then all those packages will be installed and set-up before the script is executed. You can assume that all environment variables set by the table files of the dependencies are available.

You can also assume that all tools and libraries listed as prerequisites in the install chapter of this manual are present. You may also use tools that are available on every Linux distribution, such as tar.

Rules for build scripts

- You may use the current directory to store temporary files. For example to unpack and build the source. The directory will be deleted after the installation.
- Install into $PRODUCT_DIR. The directory is created before the build script is executed.
- Do NOT make ANY modifications outside either the current directory or the directory pointed to by $PRODUCT_DIR what so ever!.

Table file

The table files serves two purposes: It lists the dependencies of the package and it contains the instructions to setup the environment before the software in the package is used.

Note that even if the table file looks a bit like a bash script on first sight, they are not. Regular bash commands will not work.

Command	Description	Examples
setupRequired	Declares a dependency. Only direct dependencies need to be listed. EUPS will traverse them recursively.	setupRequired(libpng 1.2.38+0) setupRequired(libjpeg 6b+0)
envPrepend	Adds a path in front of an environment variable. If the variable is not yet set it is set to the given path.	envPrepend(PATH, ${PRODUCT_DIR}/bin)
envAppend	Adds a path to the end of an environment variable. If the variable is not yet set it is set to the given path.	endAppend(LD_LIBRARY_PATH, ${PRODUCT_DIR}/lib))

Rules for table files

- DO NOT USE version ranges. EUPS has some support to specify version ranges with setupRequirement. DESDM does not support this. Always specify the exact version. This ensures that we build packages exactly the same way everywhere, making installations reproducible.
- Always use curly brackets around variables. EUPS will not work without them.
- Use envPrepend instead of envAppend. This is important as it ensures that we really ending up using the software in the package and not some other installation that happens to be installed on the machine.

Testing a package locally

Before adding the package to subversion (and thereby to the package repository server) it is often advantageous to test the package first on the local machine. We provide a tool that creates and installs a package locally using the same methods used to create the packages in the repository.

The tool is available on the package repository itself. Thous, before it can be used it needs to be installed (once only) and set-up:

```
$ eups distrib install eeups -t EEUPS-stable
$ setup eeups -t EEUPS-stable
```

The tool is most simple to use if both the build and table file are in the same directory and are named PRODUCTNAME.build and PRODUCTNAME.table. This is typically the case when preparing a package following the subversion directory layout described below.

From the directory containing the build and table file run

```
$ eeups_build PRODUCTNAME
```

This creates, builds and installs the package locally. The version will be 'dev+0' to indicate that this package was not installed from the eups repository server.

The command has essentially the same effect as to commit the package to subversion, wait for it to appear on the package repository server and then execute 'eups distrib install PRODUCTNAME dev+0'. Only that neither subversion nor the package repository is changed in anyway (nothing is leaving the machine). Also it is not possible to add packages with 'dev' as their package version to the eups repository server.

The tool has some other options for example to install the package in a separate place instead of the default location. You may also specify other locations of the build and table files. Use

```
$ eeups_build --help
```

To get a documentation of all options.

The eeups_build command hides the output of the build script, only printing the last few lines of the output in case of a failure. This makes it hard to track down the cause. The complete output is stored a log file in the directory:

```
$EUPS_PATH/EupsBuildDir/Linux64/[PRODUCT]/[VERSION]/[PRODUCT].log
```

This this directory is deleted if the install was successful, so the log file only exists on failure.

Subversion directory layout

All packages in the package repository originate from subversion. Thous the proper way to modify the package repository is to modify the subversion repository. The package repository will be updated to reflect any changes in subversion. It is important to stick to this layout to ensure that the packages can be located by the synchronization scripts.

The required layout builds on top of the usual trunk/tags/branches layout commonly applied in subversion. The same layout is used for products where the source code is maintained in the subversion repository and for third-party products where the source code is stored outside. In the first case the code will be stored in parallel to the build & table file. In the second the build and table files will be the only files in the directory structure.

- For each product a directory must exist in subversion. It should, but must not, be named like the product (for example 'cfitsio'). The location of the directory is not important. It has to be stored in the configuration file (see below).
- Inside this product directory a sub-directory called 'tags' is expected.
- Inside 'tags' packages will be created for every sub-directory with a name that follows the PRODUCTVERSION+PACKAGEVERSION schema described in an earlier section.
- Within the tag the build and table files are expected. They have to be named as follows:
    - PRODUCTNAME.build and PRODUCTNAME.table for packages without a configset.
    - PRODUCTNAME_CONFIGSET.build and PRODUCTNAME_CONFIGSET.table for packages with a configset
- In the special case of trunk-packages (see below) the trunk directory is also scanned and a package will be created if a build and table file are present.

Example for cfitsio

```
.../cfitsio/
            tags/
                 3.280+0/
                         cfitsio.build
                         cfitsio.table
                 3.300+0/
                         cfitsio.build
                         cfitsio.table
                 3.300+1/
                         cfitsio.build
                         cfitsio.table
```

This would create three packages with versions 3.280+0, 3.300+0 and 3.300+1.

Example for atlas

```
.../atlas/
          tags/
               3.8.4+0/
                       atlas.build
                       atlas.table
                       atlas_netlib.build
                       atlas_netlib.table
               3.8.4+1/
                       atlas.build
                       atlas.table
                       atlas_netlib.build
                       atlas_netlib.table
```

This would create four packages, two without a configset and two with configset 'netlib'. The versions would be 3.8.4+0 and 3.8.4+1.

The layout is chosen such that one can simply keep the build and table file in the base directory of the source code (directly in trunk). They will end up in the right place once the trunk is tagged. One must follow the usual rule not to make changes into an existing tag. Instead make a new tag (typically with an incremented package version).

The location of this directory tree inside the subversion repository is not important. As a convention, all products where the build & table files are not maintained in the same directory tree as the source-code are located under https://dessvn.cosmology.illinois.edu/svn/desdm/devel/eupsScripts/external. Those are mostly third-party products and internal products where we'd like to keep code and build/table separated (at least for a while).

Internal and external Products

The above structure containing the build and table files can either be stored separtely from the source or it can be the same structure in which the source code is maintained. The later implies that the development team uses the desdm subversion repository to manage the source-code. Obviously this is not the case for all products which are not developed within the consortium, such as cfitsio. We call those external products. They still have their directory tree within the desdm repository, but it will only contain the build and table files.

If the build and table files are stored in the same directory tree as the source code, we call them internal products. Internal products have the advantage that the maintainance of the build and table file becomes a natural part of the software development. Developers are encuraged to keep an up-to-date version of the build and table file in the trunk directory, changing it along with the source code if necessary. This package management system will ignore those files within the trunk since it only looks for tags. The advantage of keeping them in the trunk directory is simple: Once the software is ready for another release, just create a tag in the usual subversion way, that is by copying the trunk into a tag directory. The package management system will detect that tag, and the build and table files within, and create a new package. This is quite handy for the develpers as they don't have to do any additional effort to create a package beyond creating the subversion tag that they have to create anyway.

There are a few special applications of the above. Some products are treated like external ones, even though their source code is managed in the same subversion repository. In other words such products have two directory trees: one storing the source code and one storing the build and table files. This creates a clear separation of source-code and package management. We apply this in cases where the development team does manage the build and table files them selves. As long as the new package management system is still in development, we do this for all packages. The goal is to migrate those into 'internal' products, with just one directory tree, as soon as the new system is ready and the development team were instructed.

Another, very rare, case are packages that are not developed in the consortium, but that have the source (or binaries, if the authors don't provide the source code) in our subversion repository anyway. The package maintainer needs to fetch the code from the authors and check it in. Typically the tarball it self is stored in subversion directly. The motivation is that the usual way, uploading the tarball to our webserver, has the draw back that the webserver is publically accessible, where as our subversion repository is protected. We only do this for products where the licence does not allow public redistribution.

Register a new product

Each product needs to be registered in the configuration file. It points the scripts to the directories where it will start scanning the directory layout described above for packages.

The configuration file is stored inside subversion it self: https://dessvn.cosmology.illinois.edu/svn/desdm/devel/eupsScripts/external/eups.cfg

In the [products] section, add a line for the new product. It must have three parts separated by one or more white-spaces:

1. Name of the product
2. Product directory in subversion. This is the path that points to the directory layout described above.
3. An email address of a person that can be contacted if there is a problem with the packages from this product.

There are many products already declared which can be used as a reference.

Package testing on dedicated test machines

In DESDM we apply continuous build. Each package on the package repository is automatically build on a set of test machines. Those test machines are selected to represent the various environments where the software needs to run. This gives the developers / package maintainers an early feedback should the package fail to build on a certain hardware type or linux distribution. We are currently building up the set of test machines used for this.

The results of the continuous build are displayed here:

http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/

The current version of the webreport has a small bug: Packages will always be marked OK (green) during the time after creation and before the build servers have completed their build-test (the package status is OK during that time because no error has been reported). Once the build results come in, the status might change to ERROR (red). Always check the details of the package to see if all builds have already completed.

Remember not to change an existing EUPS tag when fixing bugs reported in the build tests. Always create a new package. It is perfectly OK to have old packages fail in a product. Most products have, or will eventually have, such obsolete, failing packages. The webreports show the packages with the highest package versions (the number after the '+').

Trunk Packages

In general the package management system follows the principle that a package is never changed once created. Trunk-packages are the exception to this rule.

A trunk package is a package for the code that is currently in the `trunk` directory of that product's subversion tree. The package is named as follows:

- Product name: `imsupport`
- Product version: `trunk`
- Package version: `+0`

Since the trunk code is volatile, so is the package. The `+0` is never incremented!

Installing Trunk Packages

Trunk packages are installed the same way as all other packages:

```
eups distrib install imsupport trunk+0
```

The difference is that the current source code from the trunk svn-directory (instead of an svn-tag) is exported, built and installed (Note: the package author has to make this true in his build script). This implies that the time when the `eups distrib` command is executed matters. If a trunk package is installed on two machines on two different days, they will very likely build with different source code.

Updating Trunk Packages

With regular packages there is no update routine required, as they never change. One just installs the newer version of the product. Since trunk packages are changing, one might have the need to update it.

We are currently planning a tool for this. But for the moment the following procedure can be used:

```
eups undeclare imsupport trunk+0
eups distrib install imsupport trunk+0
```

This causes a re-install of the package (which will use the current code from subversion). Note that this procedure does not handle dependencies. One has to manually decide which depending packages need rebuilding as well. In the case of `imsupport` one might need to apply the same procedure to `imd etrend` afterwards as well.

Creating Trunk Packages

Trunk packages are equal to regular packages in almost all aspects. A `eups.cfg` configuration file must be placed in the product's subversion directory and the `trunk_package` flag must be set:

```
.../imsupport/
             eups.cfg
             tags/
                 ...
             trunk/
               imsupport.table
               imsupport.build
```

The content of the eups.cfg must contain:

```
[product]
trunk_package=True
```

Natrually the build script of a trunk package must export the source from the trunk. It might be helpful to use the `SVN_PATH` environment variable which is available in all build scripts.

```
svn export $SVNROOT/imsupport/$SVN_PATH imsupport
cd imsupport
./configure
make
make install
```

`$SVN_PATH` is set to `trunk` for trunk packages and set to `tags/$FULL_VERSION` for regular packages. This way the build script does not need to be changed when copying the trunk to a tag directory when releasing the software.

it is highly recommended to have the source code in the same trunk directory as the build and table files. Our continuous integration scripts can only detect a change to a package if the change happens in the directory where the build and table files are stored. This is called an 'Internal Product'. See above.

Depending on Trunk Packages

Using `setupRequired` packages can depend on each other. Trunk packages can depend on other packages as they wish, but there is one important rule:

Non-trunk package must NOT depend on trunk packages. A regular package represents a very specific, unchangable version of a software. This would no longer be given if somewhere in its dependency tree a package could change. Therefore they must never depend on trunk packages or we break provenance.

Trunk packages are allowed to depend on other trunk packages as well as regular packages.

Eventually the system will enforce this rule. For the moment this is in the responsibility of the package author.

Continuous build and test

If the source of a trunk package changes, that is, someone makes a commit to the trunk, then this triggers automatically a new build and a new run of the tests.

If other packages depend on this trunk package, then those are rebuild and retested as well. This results in true continuous integration testing, where the software is automatically tested not only for itself, but also if it still works together with the other packages.

Build and test reports are listed on the package's build information page in the Web-Reports.

Creating Meta-Products

Meta-Products simplify the selection of packages. On the package repository there is an ever increasing number of products available in an ever increasing number of versions. Some products might even be available in different configurations. This makes it difficult to find the right version, especially since several versions of a product might be in use at multiple places simultaneously (production, testing, development, ...).

A meta-product is a regular product with a special use-case and a few special conventions. Which will be explained in this section.

As with regular products, meta-products have packages. Those are called meta-packages. Meta-products typically don't contain any software (but they could). The main 'content' of a meta-package are its dependencies. The dependencies of a meta-package group together a set of other packages that form a logical unit. In DESDM we have a meta-product for each pipeline. It groups together (depends on) all the software required to run this pipeline.

Meta-Products serve two main use-cases:

- Installation of all packages required to run the complete pipeline.
- Installation of a single package out of a pipeline, by specifing the product name and the pipeline name. The exact version of the package is inferred from the pipeline.

Meta-package typically have an empty build script (but the file still needs to exist) and a comparatively large table file.

All indirect dependencies (packages on which packages in the meta-package's table file depend on) are automatically part of the meta-product.

Versioning Schema

Meta-packages follow the versioning schema of normal packages. However some additional guidelines exist:

Product Name

The name of a meta-product should be all upper-case to discriminate it from normal products which should be lower-case. Typically the name of the meta-product equals to the name of the pipeline (e.g. 'FIRSTCUT').

Configset Name

Configsets are currently not used for meta-products.

Product Version

The product version should represent the status of the pipeline. It could be a number, but also a descriptive word. For example:

- 'stable'
- 'production'
- 'beta'
- 'testing'
- 'development'

It probably makes sense to agree on a common set of product version descriptors for all pipelines. We still need to do this.

Package Version

Incremented integer as with normal packages.

As with regular packages, once a package is created it it never changed. With meta-products, changes typically result from 'promoting' a software package to a certain version of a pipeline. Iin other words, the dependencies of the meta-product change. As with regular packages, this results in an incremented package version.

Marking a product as 'meta'

Meta-products follow the exact same layout in subversion. To mark a product as a meta product add a 'eups.cfg' file into the product's root directory in subversion:

```
.../FIRSTCUT/
            eups.cfg
```

```
            tags/
                stable+0/
                        cfitsio.build
                        cfitsio.table
                stable+1/
                        cfitsio.build
                        cfitsio.table
                development+100/
                        cfitsio.build
                        cfitsio.table
```

The content of the eups.cfg must be:

```
[product]
meta_product=True
```

Some remarks for those with EUPS experience

The DESDM system automatically creates a EUPS-tag for each meta-product and product version.For example there might be a tag for 'FIRSTCUT-stable'.

All packages, directly or indirectly, referenced from the meta-package with the highest package version is added to that EUPS-tag. For example the tag 'FIRSTCUT-stable' might correspond to the package 'FIRSTCUT table+5', if there is no other package with a higher package version.

The EUPS-tags are the only component in the package management system which changes over time (they are replaced when a package with a higher package version is created). This is ultimatively the reason why a user can write for example

```
eups distrib install FIRSTCUT -t FIRSTCUT-stable
```

and be sure to get the 'current' versions without explictly knowing them.

Package Life Cycle Qualifiers

Each package is given one of the following 'life cycle' qualifiers:

- OPERATIONAL (O)
- DEPRECATED (D)
- EXPERIMENTAL (E)

Based on these qualifiers, the the continuous integration build system can perform validation checks, can set tags for eups packages distributed through the distrib server. Furthermore, the webpages can be filtered for specific qualifiers - hence, help navigating in the thousands of packages we now have in the system.

Declaration of Life Cycle Qualifiers

Default qualifiers:

- Packages defined in the tags directory (of a given product) are identified as OPERATIONAL.
- Packages defined in the trunk or branches directory (of the given product) are identified as EXPERIMENTAL.

These defaults can be overruled for packages defined in the tags directory as follows:

- DEPRECATED packages can explicitly be declared in the eups.cfg file defined at product level.
- EXPERIMENTAL packages can be declared by appending an 'E' to the package version (e.g. eeups-1.4.3+4E). Note that this is not necessary for trunk and branch packages.

DEPRECATED packages can be declared in the eups.cfg file by the following syntax:

```
deprecated = 1.2.3+10; 3.2.1+0; [5.2+3,6.3+2]; [6.5+0,6.6+3); [8.8+1,*), (*,1.2+1]
```

Actually, for the ranges, a suitable ordering of the versions is adopted (the same ordering that you can see on the per product view in the eeups dashboard. Angular brackets indicate that boundary versions included, round brackets that boundary versions are excluded.

The * in [8.8+1,*) means that any version above (and including) 8.8+1 is included or, accordingly in (*,1.2+1] that any version below (and including) 1.2+1 is included.

Different versions or ranges of versions specified per qualifier are separated by a semi-colon (;).

Validation Checks

The validation checks will be based on the following rules:

- a package with qualifier O must not depend on E. Otherwise, the package will receive status=ERROR
- a package with qualifier O should not depend on D. Otherwise, the package will receive status=WARNING.

The result of these checks will be reflected in the package state and the package states are presented in the EEUPS Dashboard web pages.

Life Cycle Qualifiers as EUPS Tags

Furthermore, the qualifiers are reflected on the EUPS Distribution Server in form of according EUPS tags. It means that by eups distrib list -t 'OPERATIONAL' the operational versions can be listed. A dedicated package distribution service should serve exclusively only OPERATIONAL package versions to the DES user community.

Not yet implemented.

EEUPS Dashboard Views

The EEUPS Dashboard (see http://desbuild2.cosmology.illinois.edu/eeups/webservice/dashboard/) views can be filtered for these qualifiers (OPERATIONAL, EXPERIMENTAL, DEPRECATED, ALL). By default, the filter will be set to OPERATIONAL.

Not yet implemented.

EUPS User Utilities

User utilities support users in interacting with the EUPS system. Typical examples are utilities for creating new packages or for propagating version updates of a give package all the way up through the dependency tree.

All these tools are shipped as part of the eeups product:

```
$ eups distrib install eeups -t EEUPS-stable
$ setup eeups -t EEUPS-stable
```

or for a specific version to obtain new features in beta release status:

```
$ eups distrib install eeups 1.4.3+2
$ setup eeups 1.4.3+2
```

With the exception of trunk packages (see below), packages never change. If one releaes a new version of product, then all the other products that depend on it likely need a new + version as well, as one wishes to update its dependency. If a package 'at the bottom' of the dependency graph is released, many such new packages may have to be created. This is a time consuming and error-prone process to do manually.

Two different versions of package propagation tools exist:

- Dependency Updater (pkg_propagator): Used for updating updating a given package for version updates of a direct or indirect dependency.
- Bulk Propagator (eeups_propagator): Used for propagating version updates in the bulk of all packages.

The following rules hold true for both propagator tools:

- Existing packages are not modified.
- For each product-productversion at most one new + version is introduced, even if several changes have to be applied to it.
- Newly introduced + versions are copies of the previous + versions. The only changes are applied to the table files (`setupRequired(..)` entries) to reflect changes in the dependencies.
- No new dependencies are added, nor are dependencies removed. They may be replaced by different versions.

Dependency Updater

Running The Dependency Updater

The typical user scenario looks as follows: Create a new version of a given 'reference' package by replacing one or several of its direct or indirect dependencies by a new version.

Let's give a hypothetical example. Assume that you would like to to create a new diffimg version by replacing in diffimg-8.0.28+2 the indirect dependency python-2.7.6+1 by the updated version python-2.7.7+0. This will assume that you have already added python-2.7.7+0 into the system (which is not at the time of writing). Then you can run the following command:

```
$ pkg_propagator --replace python-2.7.6+1 --by python-2.7.7+0 --reference diffimg-8.0.28+2
```

This results in the following steps:

Step 1: Identify new package versions

The tool does not interact with a local eups repository. Hence, you don't need to first have the 'old' diffimg version installed in your environment. Rather, the tool fetches the information on new packages to be created from the webserver () and prints them to the console.

In the given example, the relevant parts in the dependency graph are

```
diffimg-8.0.28+2
--- coreUtils-0.5.2+4
     --- psycopg-2.4.6+4
          --- python-2.7.6+1
     --- cxOracle-5.1.2+8
          --- python-2.7.6+1
```

The new packages proposed by the system would be the following:

```
diffimg-8.0.28+3E
--- coreUtils-0.5.2+5E
     --- psycopg2-2.4.6+5E
     --- cxOracle-5.1.2+9E
```

where psycopg-2.4.6+5E and cxOracle-5.1.2+9E would have dependencies on python-2.7.7+0. The 'E' appended to the package version indicates that by default EXPERIMENTAL packages are created. On the console, you would see something like:

```
cxOracle-5.1.2+8             ==> cxOracle-5.1.2+9E
psycopg2-2.4.6+4            ==> psycopg2-2.4.6+5E
coreUtils-0.5.2+4              ==> coreUtils-0.5.2+5E
diffimg-8.0.28+2               ==> diffimg-8.0.28+3E
```

Note that if there would already be a psyopg2-2.4.6 version with the same dependencies as psycopg-2.4.6+4 but the new python version and identical build file there would be no need to create a new package version for psyopg2-2.4.6. This is recognized by the system and no new psyopg2-2.4.6 package version would be proposed. This means that if you specify a reference and a package to be replaced for which a propagated version already exists no propagation is performed.

Furthermore, it gives you a warning for those 'existing' packages (in the propagated graph) that have not status OK (or INCOMPLETE). Propagating packages that have already some known problem in one of its dependencies might not be a good thing to do.

Step 2: Checkout old packages and create the new packages

On the command line you are asked whether you want to proceed with the checkout. When confirming that with 'yes' the old package versions to be replaced are checked out from SVN to the current directory - unless another directory has been specified with the --workdir parameter (see below). Then, the proposed new package versions are created in the local checkout directory. Note that changes are **not** committed automatically!

Step 3: Manually check/modifiy the changes

After the checkout has completed and the changes have been applied to the working copies, the `pkg_propagator` tool exits. It does not commit the changes directly.

The user can now look at the modified working copies and inspect the work done by `pkg_propagator`. All changes are already marked with `svn add` and are ready for commit.

It is perfectly possible to make manual changes at this point. The directories are regular subversion working copies. This can be very handy, for example to manually fix the problems reported in step 1.

Step 4: Commit new packages

Commit changes by executing the commit bash script:

```
$ ./commit "New diffimg 8.0.28 package version with python updated to 2.7.7."
```

Further Options

Further command line options available for the pkg_propagator:

Parameter	
--workdir	Allows you to specify a working directory where the old package versions are checked out from SVN and new package versions will be created. Defaults to the current directory.
--operational	Once set new 'operational' package versions are proposed (without 'E' appended to the package version).

--webservice	URL to the webservice from where the information on new package versions can be obtained. Defaults to http://desbuild2.cosmology.illinois.edu/eeups/webservice/propagator . Note that this service can be called directly from a web browser. By entering a URL of the form
	
	`<webservice>/E/<replace>/<by>/<reference>`
	
	or
	
	`<webservice>/O/<replace>/<by>/<reference>`
	
	for 'experimental' or 'operational' package versions, respectively. In the angular brackets <replace>, <by>, <reference> you need to specify package versions such as python-2.7.6+1. As a result, you will see a json file with all the old ('orig') packages and the corresponding new ('prop') packages to be created.
	
	Example:
	
	`http://desbuild2.cosmology.illinois.edu/eeups/webservice/propagator/E/atlas_netlib-3.8.4+6/atlas_netlib-3.8.4+7/eeups-1.4.3+2`

As usual, use

```
$ pkg_propagator --help
```

to get help on the command line.

User Scenarios

Create New Experimental Package Version (with dependency update)

Follow the steps described above without setting the --operational parameter. With the above example:

```
$ pkg_propagator --replace python-2.7.6+1 --by python-2.7.7+0 --reference diffimg-8.0.28+2
```

This would, as described above, create the packages diffimg-8.0.28+3E, coreUtils-0.5.2+5E, psycopg2-2.4.6+5E, cxOracle-5.1.2+9E.

The package diffimg-8.0.28+3E would then be the package to test and play around with.

Create Operational Package Version associated with the Experimental Package Version

Possibly, once all the tests have successfully been concluded, you would like to create an operational version from that. Currently, the suggested way to do that is to again run the package propagator by using exactly the same parameters as above - but by setting an additional --operational flag:

```
$ pkg_propagator --replace python-2.7.6+1 --by python-2.7.7+0 --reference diffimg-8.0.28+2 --operational
```

This would propose to create new packages diffimg-8.0.28+4, coreUtils-0.5.2+6, psycopg2-2.4.6+6, cxOracle-5.1.2+10 - as long as no other package versions have been added for the given product versions (diffimg-8.0.28, coreUtils-0.5.2, psycopg2-2.4.6, cxOracle-5.1.2) in the meantime.

In the future, we may simplify that - e.g. by a command of the form

```
pkg_propagator diffimg-8.0.28+3E --operational
```

which would actually create new package versions diffimg-8.0.28+3, coreUtils-0.5.2+5, psycopg2-2.4.6+5, cxOracle-5.1.2+9 (obtained from the experimental versions by removing the 'E'). in any case, the experimental version will remain in the system.

Known Issues

- Currently, only one single dependency can be replaced at the time. Work in progress to replace multiple packages at the same time.
- No transaction handling: The propagator interacts with both, the continuous integration build system (for fetching the information which packages to create) and SVN for the checkout and the commit. The continuous build system needs some time to update its state for SVN changes (at the order of 30-60 secs) so that SVN and continuous integration build system are not guaranteed to always by concurrent. As a result, users may experience problems when submitting proposed changes - e.g. when two users want to do the same thing at the same time.

Bulk Propagator

Since the tool can easily create hundreds of packages, it is important to understand how the tool works before it is applied.

The tool operates in several steps:

Step 1: Tell eeups_propagate what to do

First the eeups_propagator analyzes the dependency graph and the arguments given to it, and decides which new packages should be created and what changes should be made their dependencies.

The following always holds:

- No existing packages are changed in any way.
- For each product-productversion at most one new + version is introduced, even if several changes have to be applied to it.
- Newly introduced + versions are a copy of the previous + version.
- The only changes made are to the `setupRequired(..)` entries in the table file.
- No new dependencies are added, nor are dependencies removed. They may be replaced by different versions.

Step 1.1: Most-recent dependencies only

The `eeups_propagator` updates the dependencies of every most-recent + version that has a dependency to a not-most-recent + version. This happens if no options are given to `eeups_propagator`:

```
$ eeups_propagator
```

For example, say the following packages exist:

```
python-2.7.3+0
python-2.7.3+1
python-2.7.3+2
pyfits-3.0.7+0   depends on python 2.7.3+0
pyfits-3.0.8+0   depends on python 2.7.3+0
pyfits-3.0.8+1   depends on python 2.7.3+1
```

Then two new packages would have to be created:

```
pyfits-3.0.7+1   depends on python-2.7.3+2
pyfits-3.0.8+2   depends on python-2.7.3+2
```

`eeups_propagator` *always* performs those changes as there is little point in having most-recent packages depend on packages with out-dated build and table files.

Such changes propagate through the dependency graph. If there is a package that depends on `pyfits-3.0.8+1` a new version of it would be created as well, pointing to `pyfits-3.0.8+2` and so on.

Step 1.2: Replace a product version

Using the `--replace oldpackage newpackage` option, `eeups_propagator` can also be used if a newly released package has not just changed its package version, but also the product version (the part before the +).

For example, say a problem with perl 5.10.1 forces us to switch to perl 5.18.1. A package `perl-5.18.1+0` is created that should replace the old `perl-5.10.1+1` which is currently used by countless other packages. By default `eeups_propagator` will not do anything here if `perl-5.10.1+1` is the most recent + version of perl.5.10.1. In this scenario we can apply the `--replace` option:

```
$ eeups_propagator --replace perl-5.10.1+1 perl-5.18.1+0
```

Now `eeups_propagator` will 'replace' all dependencies to `perl-5.10.1+1` with dependencies to `perl-5.18.1+0`, meaning that it will create a new + version for any package that depends on the old perl package.

Of course the change is propagated upwards through the dependency graph.

Step 2: Plan the changes and verify consistency

Once `eeups_propagator` is invoked, it uses the information given to it to assemble a list of all the packages to be created. It runs the same verification routines on them (locally), that would be run on the server if the packages would be created on the eups repository at this stage. This detects if the changes introduce cyclic dependencies or version conflicts of any kind.

If a problem is encountered, the list of all planned changes is printed out, together with a list of the problems found, and the user is asked if he would like to continue anyway.

if the `--askfirst` option is given, the user is asked for permission to continue even if no problems were found.

Step 3: Checking out and applying changes

`eeups_propagator` will now internally run a `svn checkout` for each affected product. The working copies are created in the local directory, unless specified differently with `--work`. It checks out the URL of the product specified in the `eups.cfg` file, that is, it checks out the directory tree under which the build and table files are stored.

The script then applies the changes as planned. It internally uses `svn copy` to create new packages and then modifies the table file as required.

The changes are **not** commited automatically!

Step 4: Manually check/modifiy the changes

After the checkout has completed and the changes have been applied to the working copies, the `eeups_propagator` tool exits. It does not commit the changes directly.

The user can now look at the modified working copies and inspect the work done by `eeups_propagator`. All changes are already marked with `svn add` and are ready for commit.

It is perfectly possible to make manual changes at this point. The directories are regular subversion working copies. This can be very handy, for example to manually fix the problems reported in step 2.

Step 5: Commit

Once sufficiently convinced that the changes are fine, they can be commited.

Since there might be a rather large number of working copies that need submitting, eeups_propagator has already prepared a bash script for this.

The script is called `commit` and is located in the same directory as the checked out working copies. Invoke it with the commit message as argument:

```
$ ./commit "Replaced perl-5.10.1+1 with perl-5.18.1+0 in all most recent \+ versions."
```

The new packages should appear in the eups repository and web pages after the usual delays.

Remember: If something goes very wrong, the changes can always be undone thanks to subversions versioning. See Red Book (scroll down to "Undoing Changes").