mancoosi
managing software complexity

Converting Eclipse metadata into CUDF

Technical Report TR5.3

Nature : Technical Report

Due date : 01/09/2010

Delivery Date: 01/09/2010

Start date of project : 01/02/2008

Duration : 36 months

Distribution: Public

Information Society Technologies

SEVENTH FRAMEWORK PROGRAMME

Web site: `www.mancoosi.org` **Blog:** `blog.mancoosi.org`

Specific Targeted Research Project
Contract no.214898
Seventh Framework Programme: FP7-ICT-2007-1

List of the authors

Project acronym	MANCOOSI
Project full title	Managing the Complexity of the Open Source Infrastructure
Project number	214898
Author list	Çagdas Bozman
Reviewers	Pietro Abate Roberto Di Cosmo
Workpackage number	WP5
Deliverable number	3
Document type	Technical Report
Version	1
Due date	01/09/2010
Actual submission date	01/09/2010
Distribution	Public
Project coordinator	Roberto Di Cosmo ⟨roberto@dicosmo.org⟩

Abstract

This technical report presents a description of the conversion of the metadata for Eclipse plugins into the Common Upgradeability Description Format (CUDF) developed by the Mancoosi Project. This work is the result of the author's internship at the PPS laboratory, Paris 7, during the summer 2010.

Contents

1 Introduction

Modern software systems are deployed in the form of a collection of components from which the user may chose the software packages that they wish to install on their platform. This choice is not permanent: additional components may need to be added, others may require updates, and obsolete or unwanted components may get removed.

In the FOSSworld, the most common platforms are the so-colled GNU/Linux *distributions*, whose software components are commonly called *packages*. It is the job of the *distribution editor* to create and maintain a coherent distribution. Each distribution editor chooses their format of *metadata* that describes some abstract properties of the packages in their distribution. The most important pieces of information in the package metadata are the name and version number of a package, its requirements, what it conflicts with, and the features it provides. In the GNU/Linux world, two families of metadata formats are most commonly used: the RPM format which stems from the Redhat distribution, and the Debian format defined by the distribution editor of the same name. There are some important differences not only between these two families of metadata formats, but also between different instances of these as defined by different distribution editors. Even when syntactic similarities might lead to the impression that the formats are more or less the same there still might be an essential difference in the *semantics* of the metadata.

One of the objectives of the Mancoosi project is to build a database of problem reports regarding failed attempts to modify the installation status of the packages on the system. For this reason, a metadata format, specifically designed to be independent of each distribution's peculiarities, has been designed: CUDF, the Common Upgradeability Description Format.

The goal of this work has been to validate the generality and expressive power of CUDF by providing an encoding into CUDF of the metadata coming from an entirely different world, the Eclipse platform with its components called *plugins*.

As we will see, it is possible to encode with some effort all the peculiarities of the Eclipse metadata into CUDF: a converter implementing this encoding has been developed and made available.

This allows, as a benefic side effect, to apply to the Eclipse world all the tools and algorithms originally developed in the Mancoosi project for the GNU/Linux distributions, and in particular an Eclipse plugin weather service tracking daily the status of Eclipse plugins has been made available.

The report is structured as follows: we briefly recall the structure of the Eclipse plugin metadata in Section 3, and the main characteristics of the CUDF format in Section 4; then we detail our translation in Section 5 and we give some examples of the information that can be presented using the Eclipse plugins Weather Service in Section 7.

2 Eclipse p2

Eclipse is a multi-language software development environment comprising and integrated development environment (IDE) and an extensible plugin system. It is written in the Java programming language and it is today used both to write software development tools and rich client applications. On important feature of eclipse from the early stage of development was its module platform to integrate and share third party components easily. The success of eclipse quickly highlight the necessity of a stable framework to handle plugins. During eclipse evolution, the system handling the plugin mechanism has evolved from a custom component able to handle hundreds of different plugins (Update Manager) to a full blow distribution platform able to handle thousand different plugins with complex interdependencies (Eclipse p2). Eclipse is based on the OSGi is a specification of a service platform where a *b*undle is a unit of modularization that is comprised of Java classes and other resources which together can provide functions to end users. As of Eclipse 3.0, the Runtime is fully based on the OSGi notion of bundle which in the rest of the document we will use as a synonym of plugin. The development of the eclipse p2 provisioning platform addressed three main problems.

The first challenge was to handle homogeneously the way OSGi and eclipse based application

were to interact with the environment The second was to provide a provisioning platform able to cope with different scenarios such as official repositories and private repositories with different release cycles The third challenge was to provide a simple solution to install and upgrade plugins with Eclipse avoiding the so called "plugin hell". The Eclipse p2 provisioning platform was released with Eclispe Galileo.

Our work is based on the infrastructure of the p2 component of Eclipse. One of the main objective of our work is to convert p2-metadata to cudf in order to facilitate its analysis with the tools and algorithms developed for the Mancoosi problem. The software itself was developed as an eclipse plugin directly using the p2 API.

3 Eclipse plugins

The components in the Eclipse platform are called *plugins* or *installable units*. Eclipse plugins are structured bundles of code and/or data that contribute functionality to the system. Functionalities can be contributed in the form of code libraries, platform extensions, or even documentation. Plug-ins define extension points, well-defined places where other plug-ins can add additional functionalities.

3.1 Eclipse Terminology

In order to make it easier for the reader, we provide a concise summary of the Eclipse terminology used in this report.

Equinox p2 is a component of the Equinox project. It provides a new provisioning system which replaces Update Manager mechanism to deal with Eclipse install, update and the installation of new functionality. This new provisioning system was introduce in the Eclipse 3.4/Ganymede release.

Installable Units (IUs) are metadatas which describe things that can be installed, updated or removed. As we say before, they are metadata, they are not things, hence they do not contain the actual *artifacts* but have essential information about them (e.g. names, version, etc). The bundle Jar is an *artifact*.

Artifacts are the actual content and may be composed of other artifacts. Bundle JARs and executable files are examples of artifacts.

Repository is a store of metadata or artifacts.

Profile are the target of install/management operations. They are a list of IUs that go together to make up a system.

3.2 Metadata of an Installable Unit for Eclipse

Eclipse installable units are composed of a list of key-values pairs. The concrete syntax used by the P2 platform is xml. In the following we give an overview of the principal fields describing an IU. An installable unit is completely determined by an identifier and a version. Other fields in the OSGi terminology describes dependencies and conflicts

Capabilities : a capability is the way for an IU to exports to other IUs what it has to offer. A capability is uniquely determined by a *namespace*, a *name* and a *version*.

Requirements : a requirement express dependencies among IUs. A requirement is represented by a *namespace*, a *name* and a *version range*[1].

[1] A version range is expressed by two number separated by a comma and surrounded by an angle bracket, meaning value included, or a parenthesis, meaning value excluded.

Dependencies among IUs must be expressed through capabilities: a particular IU may list in its requirements a set of (versioned) capabilities, that maybe fulfilled by installing one of the IUs that provides them.

A requirement can be associated to a filter (See section **??**) to enable or disable it depending on the environment where the IU will be installed. It can also be *optional* meaning the failing to satisfy the requirement does not prevent the IU from being installable.

Filter indicates in which contexts an IU can be installed. Typical filters are used to express variants of the Eclipse platform (on which OS it runs, with which GUI, etc.)

Singleton : the singleton flag specifies if it is possible to install to plugins with the same identifier at the same time.

Update : the identifier and a version range identifying predecessors to this IU. Making this relationship explicit allows us to deal with the IUs being renamed or avoid undesirable update paths.

In Figure 3.2 we provide a little example of the metadata for Eclipse installable units, showing some of the typical combinations of values one find in an Eclipse plugin repository.

```
<unit id='org.eclipse.core.net.linux.x86' version='1.0.0.I20080521'>
    ...
  <provides size='5'>
    <provided namespace='org.eclipse.equinox.p2.iu'
              name='org.eclipse.core.net.linux.x86'
              version='1.0.0.I20080521'/>
    ...
  </provides>
  <requires size='1'>
    <required namespace='osgi.bundle'
              name='org.eclipse.core.net'
              range='1.1.0'/>
    ...
  </requires>
  <filter>
    (&amp;amp; (osgi.os=linux) (osgi.arch=x86))
  </filter>
</unit>
```

Figure 1: content.xml

4 Common Upgradeability Description Format (CUDF)

The Common Upgradeability Description Format (CUDF for short) is a common format used to abstract over distribution-specific details, so that solvers can be fed with upgradeability problems coming from any supported distribution. This format is specially designed for the purpose of self-contained problem description[4].

A Cudf file is composed by three elements: a preamble, a package universe and a request.

The preamble specify global information about the CUDF document that contains it. A CUDF document must contain at most one preamble information.

A package universe contains a list of package stanza. Each stanza contains the description several facets of a package such as the package name, version, dependencies, conflicts and features declared by the package (provides).

```
package: car
version: 1
depends: engine , wheel , door , battery
installed: true

package: bicycle
version: 7

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

package: gasoline-engine
version: 2
provides: engine
conflicts: engine, gasoline-engine

package: electric-engine
version: 1
depends: solar-collector | huge-battery
provides: engine
conflicts: engine, electric-engine

[ ... ]

request: ID1
install: bicycle, electric-engine = 1
upgrade: door, wheel > 2
```

The **request** stanza describes the user upgrade request that has been submitted to the package manager. The request can be either to install, remove or upgrade a package.

5 P2 metadata translation

In this section, we show how to encode all the relevant P2 metadata into a CUDF document: in many cases, the encoding is quite natural, but there are some cases that deserve special attention, like filters, the uniqueness and the greedy flags.

We also use the output format of Debian. This format contains less information than the document CUDF, it is more portable and compact. Indeed we will convert in this format only the following information:

- the package name
- the integer version
- the dependencies
- the conflicts
- the provides
- the recommends
- the suggests

- the source version of p2 metadata

This conversion is preferred when using the tool *distcheck*. Indeed it allows us to obtain a Yaml document with the integer versions transformed in the original p2 metadata's versions, which allows for better readability. Here is an example of a Yaml document obtained from the Debian format:

```
...
 -
  package: org.eclipse.emf.rap.common.ui.source.feature.group
  version: 2.6.0.v20100914-1218
  status: broken
  reasons:
   -
    missing:
     pkg:
      package: org.eclipse.emf.rap.common.ui.source.feature.group
      version: 2.6.0.v20100914-1218
      missingdep: A.PDE.Target.Platform_Cannot_be_installed_into_the_IDE (>= 0.0.0)
...
```

Since filters are used only to distinguish among different Eclipse deployment platforms, their values actually correspond to the *architecture* specifications for GNU/Linux distributions, and we decided to follow the same approach of distributions: create a separate document for each architecture, holding the component metadata relevant for that architecture. As in GNU/Linux distribution one has different repositories for i386, hppa, arm, etc., in our treatment of Eclipse metadata we produce different documents for each combination of values found in the filters.

The enablement filter is of the form of an LDAP filter. Filters evaluation are done against a set of valued variables called an "evaluation context". Here are some examples of filters found in Helios release:

```
(osgi.os=linux) (osgi.os=win32) (osgi.os=macosx)
(&(osgi.arch=x86)(osgi.os=win32)(osgi.ws=win32))
(&(osgi.os=macosx)(osgi.ws=cocoa)(|(osgi.arch=ppc)(osgi.arch=x86)))
etc...
```

5.1 Basic metadata

5.1.1 Identifier and version

The conversion of an identifier is quite easy. We just let the same identifier in the right property which is **package** (the package name).

Like P2 the key name/version (not source version here) is unique.

5.1.2 Capabilities / Provides

A package can provides zero or more features. Installable Units advertise their features via eclipse capabilities. Capabilities are converted to CUDF as a list of pairs (name, version), where name is the encoding of the original namespace and identifier and version is the mapping of eclipse versions to CUDF versions.

For example capabilities like:

```
namespace=capns name=capn version=0.0.0
namespace=capns name=capn version=1.0.0
```

is converted into:

```
provides: capns_capn = 1, capns_capn = 2
```

5.2 Special metadata items

5.2.1 Uniqueness flag

IUs with the same identifier and with the singleton value set to false can be installed simultaneously. If the singleton flag is set to true, then only one IU with the same identifier can be installed.

This kind of constraint can be encoded in CUDF using the so called self conflicts. Then each time we will see a singleton flag set to true, we will add a self conflict.

Self conflict consist of adding the same package in its *conflicts* field. Indeed thanks to that, the package is in self conflict.

As this field indicates which packages cannot be co-installed, in any given installation, together with a given package, we will not install packages with the same identifier except himself. Hence we will have just one package with the same identifier.

Let's take an example:

$$\begin{cases} \textbf{package:} \text{ p} \\ \textbf{version:} \text{ 1} \\ \textbf{depends:} \text{ q} \geq 2, \text{r} \leq 2 \\ \textbf{conflicts:} \text{ p} \end{cases}$$

This is how we encode the self conflict.

Filter As explained above, each different filter F found in an Eclipse P2 repository is treated like:

- a suite is [here] *eclipse*
- a release is the name of different eclipse version. For example, you can have *ganymede*, *galileo*, *helios*, etc.
- an architecture is the same as in Debian. For example you have *x86*, *s390*, *ppc*, etc.

We produce a different CUDF document D_F for each suite/release/architecture.

Hence, when we find a requirement R with a filter F, we add the requirement to the metadata for the document D_F, and we ignore it for all other documents.

For example, if the following requirement is found for an IU u,

`(os=linux)` $\rightarrow$ `req`

the encoding of *req* will be added to the translation of u only in the document corresponding to the architecture `os=linux`.

Greedy It is a very different concept compared to what already exist in CUDF. It was added to control the addition of IUs as part of the potential IUs to install in order to satisfy a request. In other term, if greedy is set to true for a given IU, then we added all IUs in a pool of potential candidats which can satisfy the dependency of this IU. Else we just wait that other dependencies (its dependencies or others) bring in what is necessary for its satisfaction.

For more detailed please see the document written by Pascal Rapicault and Daniel Le Berre: "*Dependency Management for the Eclipse Ecosystem: An Update*" [3].

To convert these non greedy requirements, we have three cases:

- First case regroup actually two cases: when optional value is set to *false* (it does not matter of the value of greedy). In this case this is a strong requirement (dependency). Then we use the *depends* field.
- Then when optional value and greedy value are both set to *true*, this is an optional case. Then we use the *recommends* 5.4 field.

- Finaly when the optional value is set to *true* and greedy value is set to *false*. This is the weakest requirement. Then we use the *suggests* 5.4 field.

5.3 P2 comparison function and CUDF version mapping

As we said previously, a version identifiers have four components:

- a major version
- a minor version
- a micro version
- a qualifier (optional)

Let's see some examples:

```
0.0.0 < 0.0.1 < 0.1.0 < 1.0.0 < 2.1.0.qualifier < 3.0.0.azerty
```

We can see the comparison function in org.osgi.framework.Version [2]:

```
public int compareTo(Version v)
```

According to java documention, a version A is less than a version B if the major component of A is less than B major version or if both of major components' are equals, then we do the same thing with the minor and the micro version and if the three are equals then we compare the qualifier component with the *String.compareTo()* [1] function. See documention for more information [2].

We produce a mapping from versions in Eclipse P2 metadata to CUDF in the standard way suggested for CUDF: we just order all the version of a component according to the Eclipse comparison function, and then we replace each version by its position in this ordering.

For example, consider the following metadata:

id: mancoosiIu version: 0.0.1

Capabilities:

```
namespace=a name=A version=0.0.0
namespace=a name=A version=1.0.0
namespace=b name=B version=2.3.4
```

Requirements:

```
namespace=b name=B range=[1.0.0, 3.0.0)
namespace=c name=C range=[1.0.0.azerty, 2.6.0]
namespace=c name=C range=[2.0.0, 4.0.0) optional=true
```

Then the version mapping to CUDF is as follows:

$$\text{A} \begin{cases} \texttt{0.0.0} \rightarrow 1 \\ \texttt{1.0.0} \rightarrow 2 \end{cases} \quad \text{B} \begin{cases} \texttt{1.0.0} \rightarrow 1 \\ \texttt{2.3.4} \rightarrow 2 \\ \texttt{3.0.0} \rightarrow 3 \end{cases} \quad \text{C} \begin{cases} \texttt{1.0.0.azerty} \rightarrow 1 \\ \texttt{2.0.0} \rightarrow 2 \\ \texttt{2.6.0} \rightarrow 3 \\ \texttt{4.0.0} \rightarrow 4 \end{cases}$$

After the conversion, we have:

$$\begin{cases} \textbf{package:} \text{ mancoosiIu} \\ \textbf{version:} \text{ 1} \\ \textbf{provides:} \text{ aA} = 1, \text{aA} = 2, \text{bB} = 2 \\ \textbf{depends:} \text{ bB} \geq 1, \text{bB} < 3, \text{cC} \geq 1, \text{cC} \leq 3 \\ \textbf{recommends:} \text{ cC} \geq 2, \text{cC} < 4 \\ \textbf{conflicts:} \text{ mancoosiIu} \quad \textbf{only if singleton flag is set to true} \\ \textbf{eclipselastversion:} \text{ true} \\ \textbf{eclipsesourceversion:} \text{ 0.0.1} \end{cases}$$

5.4 Extra properties

We also add to the CUDF document a set of extra properties, that are useful for maintaining some additional information about the Eclipse P2 components in the CUDF translation.

eclipsesourceversion In order to maintain a mapping between P2 and CUDF versions, for each CUDF package we add an extra property named **eclipsesourceversion** of type *string* that contains the original P2 source version of the IU.

recommends Dependencies which are optional. The solver will do the best to consider this field. This property's type is *vpkgformula* [4].

suggests When the optional flag is set to *true* and the greedy flag to *false*, the dependency is add to the suggests field. This property's type is *vpkgformula* [4].

eclipsegreedydeps Only greedy 5.2.1 dependencies are in this field. This property's type is *vpkgformula* [4].

eclipselastversion This property's type is *boolean*. For an IU with differents versions, this flag allows to know which one is the last version of the set.

eclipsecategory For each IU we add an extra property named **eclipsecategory** of type *boolean* that contains the category status of an IU. If an IU is a category then this field is set to *true*.

eclipsegroup For each IU we add an extra property named **eclipsegroup** of type *boolean*. If an IU is a group then this field is set to *true*.

eclipsefragment For each IU we add an extra property named **eclipsefragment** of type *boolean*. If an IU is a fragment then this field is set to *true*.

eclipsepatch For each IU we add an extra property named **eclipsepatch** of type *boolean*. If an IU is a patch then this field is set to *true*.

6 Using the converter

6.1 Configuration file

To start converting some IUs, you will need to write a configuration file. The concrete syntax of the file is the ini format [**?**]. Here are the differents sections of this configuration file.

Comments: same as ini files (';' and '#')

Release: it is a string identifier of the release like galileo, ganymede, helios, ...

Context: the key/value 's have to be like `n= property1:value1, ..., propertyN:valueN` where n is a integer and N $\geq$ 1

Baseline: the key/value 's have to be like `baseline= repos1, ..., repoN` where x is an ID and N $\geq$ 1

Main: the only key/value has to be exactly like `main= main_repository`

Options contains all optional property describe below.

Checkonly: the key/value 's have to be like `checkonly= package1, ..., packageN` where x is an integer and N $\geq$ 1

Outdir: the path to the output directory. This field is optional.

Format: this is the output files' format. The value is *deb* and *cudf*.

Distcheck: this is the path to distcheck launcher.

Example:

```
## This is a comment :-)
[context]
baseline=http://download.eclipse.org/releases/helios/
main=http://download.eclipse.org/releases/helios/
release=helios
1=osgi.os:win32, osgi.ws:win32, osgi.arch:x86
2=osgi.os:linux, osgi.ws:gtk

[options]
checkout=no
outdir=/path/to/my/outputdirectory/
format=deb
```

6.2 Filtering

As we said above, we filtered each document per context. The most used filters are illustrated in the figure below 2.

os :	linux	win32	macosx	hpux	solaris	aix	etc.
arch:	x86	x86_64	ppc	ppc64	sparc	etc.	
ws:	gtk	win32	carbon	cocoa	motif	etc.	

Figure 2: Most common filters in p2 metadata

6.3 Translation algorithm

The translation algorithm is as follows.

First before starting the translation, we create a context and the document that will be create, will contain only the package which are on the context. Thanks to that there is no need to do something special to deal with filters. Then we continue in converting the *Preamble* stanza. In this first step we add all new properties like *eclipsesourceversion*,*recommends*, *suggests*, etc. with the right types (*boolean*, *vpkgformula*, etc.).

Secondly we convert all packages (IUs). Before doing this, we check if the IU is really in the context created above. If the filter pass then we start the real conversion, i.e. conversion of the package name, the integer version, the source version, the self-conflict case, etc. We fill all the optional and not optional properties described above. Each IU and requirements have filters, so we use the same function which checks if the IUs or the requirements are on the context. If they are not, we just ignore them. Otherwise we start the translation. To deal with the integer versio, we have a special case. Before starting the conversion we scan all the IUs and we associate the right integer to each IU by ordering them with the comparison function 5.3.

As last step we finish by converting the request stanza if needed.

7 Weather report

One application we have wanted to use was the weather report. Indeed we inspired that application from debian weather report (http://edos.debian.net/weather/).

The "weather" of a given Debian-based distribution is an indication of how safe it is on a given day to attempt some package installation/upgrade. A "bad day" is a day in which a sensible percentage of that distribution repository is not installable due to unsatisfiable inter-package dependencies.

Then Eclipse weather is intended to indicate the state of plugins for every version of Eclipse. Indeed, we differentiate these different contexts for the different version of eclipse:

- Operating System (os)
- Window Screeners (ws)
- Architecture (arch)

First we use the cudf converter to generate a cudf document. Then we use the mancoosi tools: "distcheck".

After distcheck finished, a Yaml document is generate. This document contains all the necessary information to check the state of plugins for a version of eclipse. This document is then entered into the sqlite database that we use for the weather report. In fact the database contains the list of documents and the list of packages with the reasons for non installability. This step can be performed automatically using a cron.

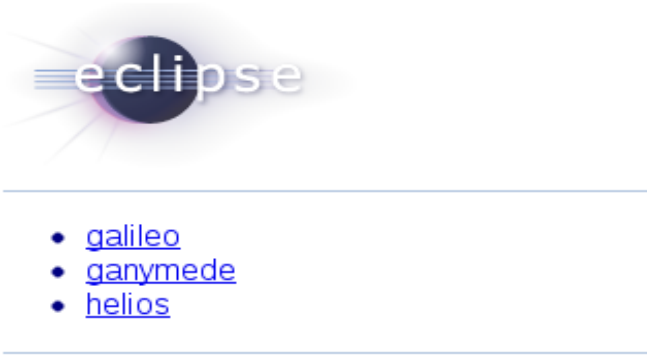


Figure 3: General view of weather report

Once the database completed, the pages are written with Django and are generated automatically. We will allow the search for os and/or ws and/or arch. We also will allow the ability to

search by date. Indeed, as each document is unique it will be easy to identify and retrieve each document by date. Here is an example of Eclipse helios:

Figure 4: Example for Helios

the list of different architecture:

Uninstallable Packages in Debian helios / linux

Clicking on the numbers in the table gives a detailed listing.

Page 1 of 1

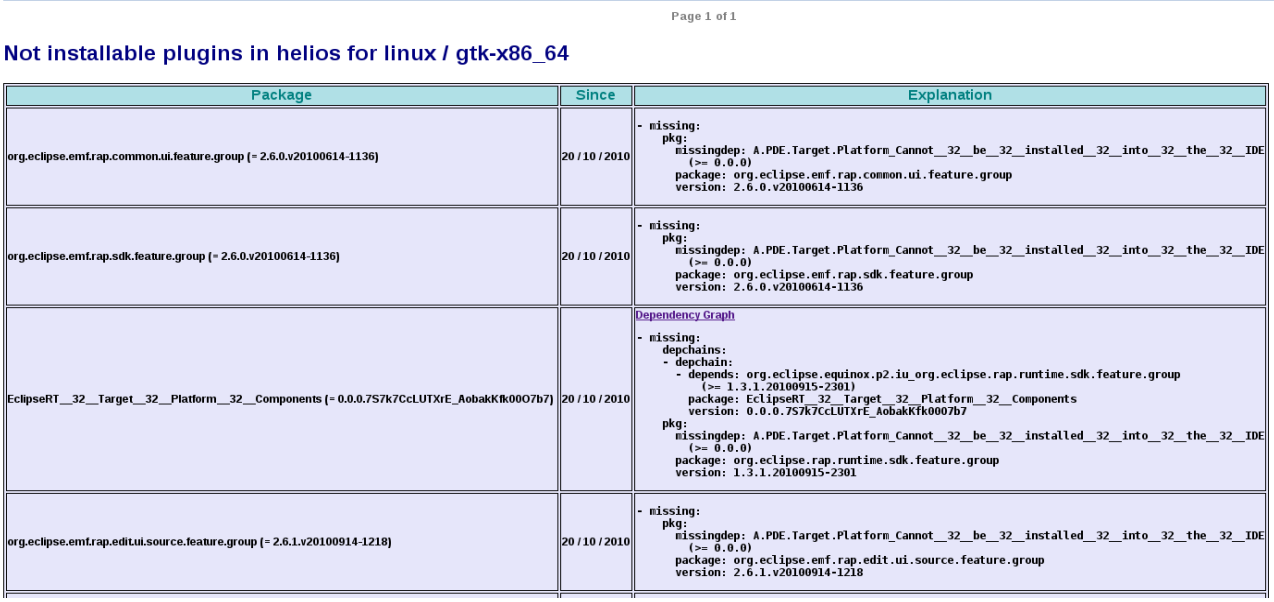
Date	gtk-ppc	gtk-ppc64	gtk-s390	gtk-s390x	gtk-x86	gtk-x86_64	motif-x86
Oct. 20, 2010, 3:20 p.m.	18	18	18	18	18	18	18
	n/a	n/a	n/a	n/a	n/a	n/a	n/a

No releases found.

Page 1 of 1

Figure 5: Detailed view of a document

and overview of detailed view of a document:



Page 1 of 1

Not installable plugins in helios for linux / gtk-x86_64

Package	Since	Explanation
org.eclipse.emf.rap.common.ui.feature.group (= 2.6.0.v20100614-1136)	20 / 10 / 2010	- missing: pkg: missingdep: A.PDE.Target.Platform_Cannot__32__be__32__installed__32__into__32__the__32__IDE (>= 0.0.0) package: org.eclipse.emf.rap.common.ui.feature.group version: 2.6.0.v20100614-1136
org.eclipse.emf.rap.sdk.feature.group (= 2.6.0.v20100614-1136)	20 / 10 / 2010	- missing: pkg: missingdep: A.PDE.Target.Platform_Cannot__32__be__32__installed__32__into__32__the__32__IDE (>= 0.0.0) package: org.eclipse.emf.rap.sdk.feature.group version: 2.6.0.v20100614-1136
EclipseRT__32__Target__32__Platform__32__Components (= 0.0.0.7S7k7CcLUTXrE_AobakKfk00O7b7)	20 / 10 / 2010	Dependency Graph - missing: depchains: - depchain: - depends: org.eclipse.equinox.p2.iu_org.eclipse.rap.runtime.sdk.feature.group (>= 1.3.1.20100915-2301) package: EclipseRT__32__Target__32__Platform__32__Components version: 0.0.0.7S7k7CcLUTXrE_AobakKfk00O7b7 pkg: missingdep: A.PDE.Target.Platform_Cannot__32__be__32__installed__32__into__32__the__32__IDE (>= 0.0.0) package: org.eclipse.rap.runtime.sdk.feature.group version: 1.3.1.20100915-2301
org.eclipse.emf.rap.edit.ui.source.feature.group (= 2.6.1.v20100914-1218)	20 / 10 / 2010	- missing: pkg: missingdep: A.PDE.Target.Platform_Cannot__32__be__32__installed__32__into__32__the__32__IDE (>= 0.0.0) package: org.eclipse.emf.rap.edit.ui.source.feature.group version: 2.6.1.v20100914-1218

Figure 6: Detailed view of a document

Furthermore with Pydot, we suggest where possible, a png image of the path which is the cause of non installability of the package(Figure 4). It will be easier to visualize the path that can be sometimes quite long.

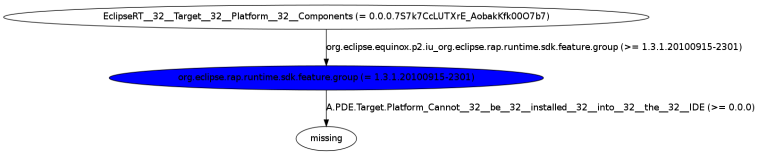


Figure 7: Detailed view of a package (dot file)

8 Conclusion

To conclude this project has been a great experience. We could use Mancoosi tools in a different context: Java world with Eclipse and its plugins.

One difficulty has been to limit the dependencies to Eclipse. Of course it always remain a dependency necessary for the proper functioning of cudf converter.

Once the cudf document obtained, we could have interesting results for eclipse plugins. Indeed it can be very interesting for Eclipse developers to check the states of plugins installability and at each stage of their development.

Finally we hope that the weather report will be very useful to Eclipse developers, perhaps even use in the Eclipse market.

9 Acknowledgements

Project coordinator

- Roberto Di Cosmo <roberto@dicosmo.org>

Mancoosi team

- Pietro Abate <pietro.abate@pps.jussieu.fr>
- Stefano Zacchiroli <zack@pps.jussieu.fr>
- Jérôme Vouillon <jerome.vouillon@pps.jussieu.fr>
- Ralf Treinen <ralf.treinen@pps.jussieu.fr>
- Jaap Boender <jaap.boender@pps.jussieu.fr>

Eclipse P2 team

- Pascal Rapicault <pascal@sonatype.com>

References

[1] Oracle. *Class String*. http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html.

[2] OSGi Service Platform. *Class Version*. http://www.osgi.org/javadoc/r4v42/org/osgi/framework/Version.html.

[3] P. Rapicault and D. L. Berre. Dependency management for the eclipse ecosystem: An update. Technical report.

[4] R. Treinen and S. Zacchiroli. Common upgradeability description format (cudf) 2.0. Technical report, 2009.