Upgrade description formats: generalities and DUDF submission format

Ralf Treinen
Stefano Zacchiroli

Technical Report 001

Version 2.0

24 November 2009

Abstract

The solver competition which will be organized by Mancoosi relies on the standardized format for describing upgradeability problems. This document describes the layout of the infrastructure for building a data base of upgradeability problems, and in particular the DUDF format (Distribution Upgradeability Description Format) which is intended for the submission of upgrade problem instances from user machines to a (distribution-specific) database of upgrade problems.

Status of this Document

The contents of this document in version 2.0 is based on the chapters of Deliverable D5.1 [TZ08] on DUDF, with some minor modifications (see Chapter C). Future modifications of DUDF shall be documented as new versions of the current document.

Conformance

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [Bra97].

Contents

Chapter 1

Introduction

The aim of work package 5 (WP5) of the Mancoosi project is to organize a solver competition to attract the attention of researchers and practitioners to the upgrade problem[1] as it is faced by users of FOSS distributions [DC08]. The competition will be run by executing solvers submitted by the participants on upgrade problem descriptions (or "problems", for short) stored in upgradeability problem data bases (UPDBs). A substantial part of the problems forming UPDBs, if not all of them, will be real problems harvested on user machines; users will be given tools to submit on a voluntary basis problems to help Mancoosi assemble UPDBs.

In such a scenario, problem descriptions need to be saved on filesystems (for long term storage) and transmitted over the network (to let them flow from user machines to UPDBs). This document gives the specifications of document formats used to represent problem instances in the various stages of their lives.

1.1 Two different upgrade description formats

Upgrade description formats serve at least two different purposes:

Problem submission problems will be created on distant user machines and need to flow to more centralized UPDBs. Both the user machine itself and the network connection may have only limited resources.

Problem description problems will be stored by Mancoosi to form a corpus of problems on which the solvers taking part in the competition will be run.

In the Mancoosi Description of Work we announced the definition of a so-called *Common Upgradeability Description Format*, abbreviated CUDF, that would serve these two purposes. It turned out that having one single format for both purposes is not practical since both purposes come with contradicting constraints: problem submissions should take as few resources as possible on a user's machine, and they may contain references that are meaningful only in

[1] Throughout this specification, the word "*problem*"—as in "upgrade *problem*"—is used in the sense of "problem solving". Hence, an "upgrade problem" is just an upgrade *scenario* in which a solution to an upgrade request posed by a user needs to be found. In particular, an upgrade problem is not necessarily *troublesome* for users: the whole upgrade process can go well; still, in its evolution, it has posed an upgrade problem (in the sense of this specification), that a software entity has solved, most likely finding a suitable upgrade path.

the context of a particular distribution. On the other hand, problem descriptions as used for the competition are not subject to strong resource limitations but must be self-contained and must have a formally defined semantics that is independent from any particular distribution.

As a consequence, we decided to define two different formats, one for each of the main purpose:

DUDF (Distribution Upgradeability Description Format) This is the format used to submit a single problem from user machines to a UPDB. DUDF is specialized for the purpose of problem submission.

DUDF instances (or "DUDFs" for short) need to be as compact as possible in order to avoid inhibiting submissions due to excessive bandwidth requirements. To this end, the DUDF specification exploits distribution-specific information, such as the knowledge of where distribution-wide metadata are stored and where metadata about old packages can be retrieved from mirrors that may or may not be specific to Mancoosi.

Since a DUDF is by its very nature distribution dependent there cannot be a a single complete DUDF specification. We rather present in Chapter 2 a generic specification of DUDF documents, the *DUDF skeleton*, which has to be instantiated to a full specification by all participating distributions. Documents to be published separately, one per distribution, will describe how the general scheme is instantiated by the various distributions.

All in all we have a *family of DUDF specification instances*: Debian-DUDF, RPM-DUDF, etc.; one for each possible way of filling the holes of the generic DUDF specification. How many instances should be part of the DUDF family? We recommend to have one instance for each distribution taking part in the competition. While different distributions may share a common packaging format, they may also allow for different means of compact representations, for example due to the different availability of mirrors with historical information. Furthermore, there are sometimes subtle semantic differences from distribution to distribution, hidden behind a shared syntax. To discriminate among different distributions, an appropriate distribution information item is provided. Of course, nothing prohibits different distributions to agree upon the same DUDF specification instance in case they find that this is feasible.

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

The conversion from a given DUDF to CUDF expands the compact representations that have been performed for the purpose of submission, exploiting distribution-specific knowledge. At the end of such a conversion, a problem described in CUDF is self-contained, only relying on the defined semantics of an upgradeability problem, which includes the starting state, the user query, and probably non-functional quality criteria.

Structure of this document This document is structured as follows: Chapter 1 gives introductory information about the various kinds of documents involved in the organization of the competition and about the problem submission infrastructure. Chapter 2 contains the actual specification of the DUDF skeleton; this chapter is normative and defines what it takes for a document to be valid with respect to its specification. Appendixes to this document contain various non-normative information, which may be helpful to implementors of DUDF. Documents to be made available separately will describe how each distribution is instantiating the DUDF skeleton.

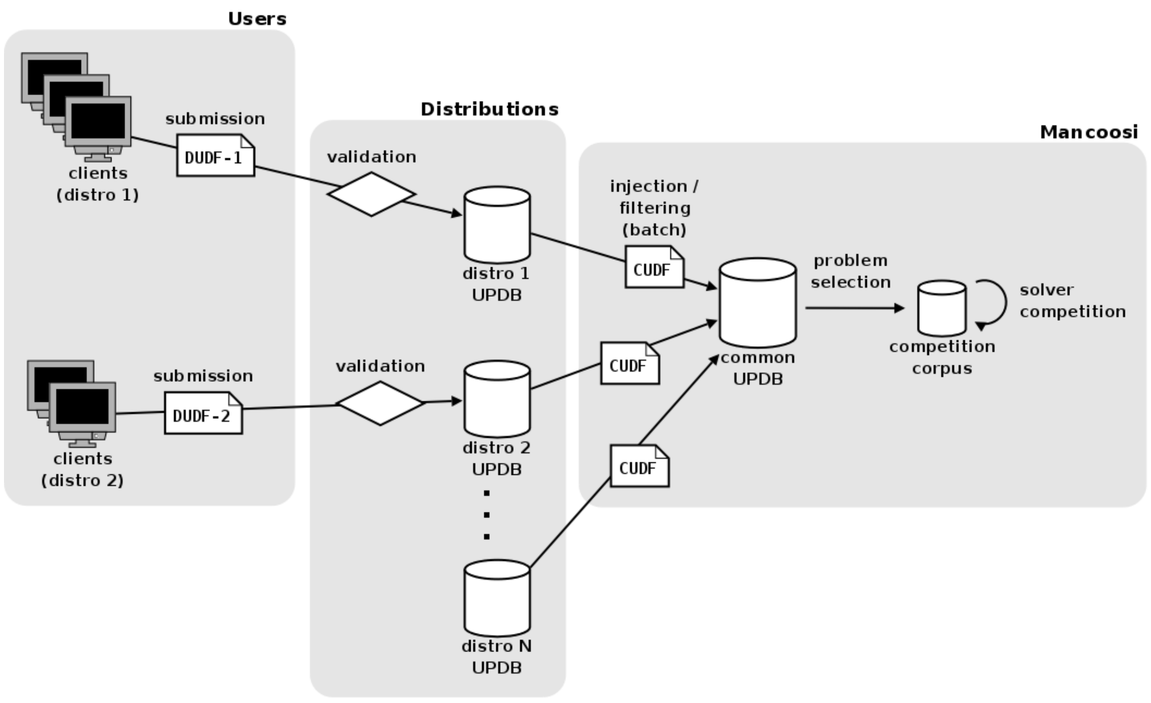

Figure 1.1: Data flow of UPDB submissions, from users to the corpus of problems for the competition

1.2 Problem data flow and submission architecture

Figure 1.1 gives an overview of the data flow of upgrade problems from user machines to the actual solver competition; several stages of transmission and filtering, as well as several different formats are involved.

Problems originate on user machines and are serialized in DUDF format (i.e. distribution-specific DUDF instances) using some client software. DUDF documents created that way will then be submitted to distribution-specific repositories using some other client software. All involved client software will be provided by distributions, such software will constitute implementations of the DUDF specification.

Distributions need to set up their own repositories to collect DUDF submissions coming from their users. Submissions that do not match the minimal quality requirements of DUDF will be rejected during a validation phase; this mainly boils down to rejecting problems that are not reproducible, see Chapter 2 for more details. All submissions that survive the validation phase are stored by the distribution editor in a distribution-specific UPDB.

Periodically, problems collected by distributions will be injected into a common (i.e. distribution-independent) UPDB, hosted on an infrastructure provided by Mancoosi as a project resource. The injection happens in CUDF format since distribution-specific details are not useful for the purpose of running the competition. Distributions are in charge of performing the conversion from DUDF to CUDF as they are the authoritative entities for the semantics of their proper DUDF instance and for resolving distribution-specific references. When exactly the conversion is performed is not relevant as long as CUDFs are ready to be injected when the periodic injections take place.

Among all the problems collected in the common UPDB, a subset of "interesting" problems will then be selected to form a corpus of problems on which the competition will be run. The act of selecting problems will not change the document format: the resulting corpus will still be a set of CUDF documents, chosen as a subset of the common UPDB.

1.3 Glossary

This section contains a glossary of essential terms which are used throughout this specification.

Distribution A collection of software packages that are designed to be installed on a common software platform. Distributions may come in different flavors, and the set of available software packages generally varies over time. Examples of distributions are Mandriva, Caixa Mágica, Pixart, Fedora or Debian, which all provide software packages for the the GNU/Linux platform (and probably others). The term *distribution* is used to denote both a collection of software packages, such as the *lenny* distribution of Debian, and the entity that produces and publishes such a collection, such as Mandriva, Caixa Mágica or Pixart. The latter are sometimes also referred to as *distribution editors*.

Still, the notion of distribution is not necessarily bound to FOSS package distributions, other platforms (e.g. Eclipse plugins, LaTeX packages, Perl packages, etc.) have similar distributions, similar problems, and can have their upgrade problems encoded in CUDF.

Installer The software tool actually responsible for physically installing (or de-installing) a package on a machine. This task particularly consists in unpacking files that come as an archive bundle, installing them on the user machine in persistent memory, probably executing configuration programs specific to that package, and updating the global system information on the user machine. Downloading packages and resolving dependencies between packages are in general beyond the scope of the installer. For instance, the installer of the Debian distribution is `dpkg`, while the installer used in the RPM family of distributions is `rpm`.

Meta-installer The software tool responsible for organizing a user request to modify the collection of installed packages. This particularly involves determining the secondary actions that are necessary to satisfy a user request to install or de-install packages. To this end, a package system allows to declare relations between packages such as dependencies or conflicts. The meta-installer is also responsible for downloading necessary packages. Examples of meta-installers are `apt-get`, `aptitude` and `URPMi`.

Package A bundle of software artifacts that may be installed on a machine as an atomic unit, i.e. packages define the granularity at which software can be added to or removed from machines. A package typically contains an archive of files to be installed on a machine, programs to be executed at various stages of the installation or de-installation of a package, and metadata.

Package status A set of metadata maintained by the installer about packages currently installed on a machine. The package status is used by the installer as a model of the software installed on a machine and kept up to date upon package installation and removal. The kind of metadata stored for each package varies from distribution to distribution, but typically comprises package identifiers (usually name and version), human-oriented information such as a description of what the package contains and a formal declaration of the inter-package relationships of a package. Inter-package relationships can usually state package requirements (which packages are needed for a given one to work properly) and conflicts (which packages cannot coexist with a given one).

Package universe The collection of packages known to the meta-installer in addition to those already known to the installer, which are stored in the package status. Packages belonging to the package universe are not necessarily available on the local machine—while those belonging to the package status usually are—but are accessible in some way, for example via download from remote package repositories.

Upgrade request A request to alter the package status issued by a user (typically the system administrator) using a meta-installer. The expressiveness of the request language varies with the meta-installer, but typically enables requiring the installation of packages which were not previously installed, the removal of currently installed packages, and the upgrade to newer version of packages currently installed.

Upgrade problem The situation in which a user submits an upgrade request, or any abstract representation of such a situation. The representation includes all the information needed to recreate the situation elsewhere, at the very minimum they are: package status, package universe and upgrade request. Note that, in spite of its name, an upgrade problem is not necessarily related to a request to "upgrade" one or more packages to newer versions, but may also be a request to install or remove packages. Both DUDF and CUDF documents are meant to encode upgrade problems for different purposes.

Chapter 2

Distribution Upgradeability Description Formats

This chapter contains the specification of the Distribution Upgradeability Description Formats (DUDFs). Their purpose is to encode upgrade problems as faced by users, so that they can be submitted as candidate problems for the solver competition organized by the Mancoosi project.

Additionally, DUDF can also be used as a format to store information about the execution of a meta-installer on a user machine. A possible use case for this is to trace information for the purpose of composing problem reports against meta-installers. This is an added benefit for distribution editors which is, however, beyond the scope of the Mancoosi project itself.

Technically, the DUDF specification is not complete, in the sense that some parts of DUDF documents are under-specified and called "holes". How to fill in those holes is a distribution-specific decision to be taken by each distribution implementing DUDF. The overall structure of DUDF documents is defined by the current document and is called the *DUDF skeleton*.

2.1 Upgrade problems

Upgrade problems manifest themselves at each attempt to change the package status of a given machine using a meta-installer. One of the aims of WP5 for the solver competition is to collect *upgrade problem descriptions* which faithfully describe the upgrade problems faced by users when invoking a meta-installer on their machine. Informally, "faithfully" means that the descriptions should contain all information needed to reproduce the problem reported by the user, and possibly to find better solutions if they exist.

As discussed in Chapter 1, problem descriptions will be encoded as DUDFs and submitted to distribution-specific repositories. Two kinds of submissions are supported by DUDF:

(a) Sole problem descriptions.

(b) Pairs ⟨*problem description*, *problem outcome*⟩ where the outcome is a representation of the actual result of the originating meta-installer which has been used to generate the problem.

Pairs problem/outcome are the kind of submissions to be used for the competition. Their validity as submissions can be checked by attempting to reproduce them upon receipt (see below), and the outcome of competing solvers can be compared not only among each other, but also with respect to the originating meta-installers in order to check whether they are doing better or worse than the contenders.

Sole problem descriptions cannot be checked for reproducibility. As such they are not interesting for the competition since they can not be "trusted". Still they can be useful for purposes other than the competition. In particular they can be used—as well as pairs problem/outcome—by users to submit bug reports related to installers, meta-installers, and also incoherences in package repositories [EDO06]. This intended use is the main reason for supporting them in this specification.

2.2 Content

A DUDF document consists of a set of *information items*. Each item describes a part of the upgrade problem faced by the user. In this section we list the information items (or *sections*) that constitute the different kinds of DUDF submissions.

The actual format and content of each information item can either be fully described by this specification, or be specific to some of its instances (and hence not described here). In the latter case, we distinguish among parts which are specific to the *installer* and parts which are specific to the *meta-installer*. Installer-specific parts have content and format determined by the installer (e.g. rpm, dpkg, etc.) in use; similarly, parts specific to the meta-installer are determined by the meta-installer (e.g. apt-get, URPMi, etc.) in use.

Unless otherwise stated, all information items are required parts of DUDF documents.

The submission of a *sole upgrade problem description* consists of the following information items:

Package status (i.e. *installer status*) the status of packages currently installed on the user machine.

This item is installer-specific, but can also contain data specific to the meta-installer in case the meta-installers save some extended information about local packages. A concrete example of such extended information is the manual/automatic flag on package installation used by `aptitude` on Debian to implement "garbage collection" of removed packages.

Package universe the set of all packages which are known to the meta-installer, and are hence available for installation. This item is specific to the meta-installer.

The package universe is composed of one or more *package lists*; a number of well-known formats do exist to encode package lists. The package universe can generally be composed of several package lists, each encoded in a different format. Each package list must be annotated with a unique identifier describing which format has been used to encode the package list. A separate document will be published to list the set of well-known package list formats, as well as their unique identifiers.

Requested action the modification to the local package status requested by the user (e.g. "install X", "upgrade Y", "remove Z"). This item is specific to the meta-installer.

Desiderata user preferences to discriminate among possible alternative solutions (e.g. "minimize download side", or "do not install experimental packages"). The exact list of possible user preferences depends on the distribution, and on the capabilities of the meta-installer (for instance, for Debian's `apt` these may be defined in the file `/etc/apt/preferences`).

This information item is optional.

Tool identifiers two pairs ⟨*name*, *version*⟩ uniquely identifying the installer and meta-installer which are in use, in the context of a given distribution. One pair identifies the *installer* used, the other the *meta-installer* used.

Distribution identifier a string uniquely identifying the distribution run by the user (e.g. `debian`, `mandriva`, `pixart`, . . .), among all the implementations of DUDF.

As far as GNU/Linux distributions are concerned, a hint about what to use as a distribution identifier comes from the file `/etc/issue`. Its content should be used as distribution identifier where possible.

Timestamp a timestamp (containing the same information encoded by dates in RFC822 [Cro82] format, i.e. the same as used in emails) to record when the upgrade problem has been generated.

Problem identifier (i.e. *uid*) a string used to identify this problem submission *univocally*, among other submissions sent to the same distribution.

The intended usage of this information item is to let CUDF documents cross-reference the DUDF documents which were used to generate them.

In addition to what is stated above, the submission of a pair problem/outcome also contains the following information items:

Outcome either the new local package status as seen by the used meta-installers (in case of *success*) or an error message (in case of *failure*, i.e. the meta-installer was not able to fulfill the user request). The error message format is specific of the used meta-installer, it can range from a free-text error message to a structured error description (e.g. to point out that the requested action cannot be satisfied since a given package is not available in the package universe).

It is worth noting that Mancoosi is not interested in all kinds of errors, and that not all errors reported to the end user mean a failure that is interesting for the competition. Mancoosi is interested only in errors stemming from the resolution of package relations, which is the case when the meta-installer is not able to solve the various constraints expressed in the summary information *about* the packages. Mancoosi Workpackage 5 is not interested in runtime errors such as installation failures due to disks running out of space or execution errors of maintainer scripts. These errors, however, may still be relevant for submitting problem reports to a distribution vendor using the DUDF format.

Note that tool identifiers are part of the problem description since the requested action depends on the tools the user is using. Since available actions, as well as their semantics, can change from version to version, tool versions are also part of the problem description.

The distribution identifier is needed to avoid bloating the number of specified DUDFs too much. We observe that similar distributions (e.g. Debian and Ubuntu) can submit upgrade problems using the very same submission format (say Debian-DUDF). However, even though extensional data (see Section 2.3) are independent of which of the similar distributions were used, intensional data are not. Indeed, there is no guarantee that package p at version v is the same on Debian and Ubuntu; similarly there is no guarantee that an intensional package universe reference originated on Debian is resolvable using Ubuntu historical mirrors and vice-versa. Using the

- dudf:
    - version: 2.0
    - timestamp: *timestamp*
    - uid: *unique problem identifier*
    - distribution: *distribution identifier*
    - installer:
        - name: *installer name*
        - version: *installer version*
    - meta-installer:
        - name: *meta-installer name*
        - version: *meta-installer version*
    - problem:
        - package-status:
            - installer: $\boxed{\textit{installer package status}}$
            - meta-installer: $\boxed{\textit{meta-installer package status}}$
        - package-universe:
            - package-list$_1$ (format: *format id.*; filename: *path*; url: *url*): $\boxed{\textit{package list}}$
            - . . .
            - package-list$_n$ (format: *format identifier*; filename: *path*): $\boxed{\textit{package list}}$
        - action: $\boxed{\textit{requested meta-installer action}}$
        - desiderata: $\boxed{\textit{meta-installer desiderata}}$
    - outcome (result: *one of "success", "failure"*):
        - error: $\boxed{\textit{error description}}$ (only if result is "failure")
        - package-status: (only if result is "success")
            - installer: $\boxed{\textit{new installer package status}}$
            - meta-installer: $\boxed{\textit{new meta-installer package status}}$
    - comment: $\boxed{\textit{additional, user-provided information}}$ (optional)

Figure 2.1: The DUDF skeleton: information items and holes corresponding to problem/outcome submissions.

distribution identifier we can reuse the same DUDF instance for a set of similar distributions since the distribution identifier allows us to resolve the ambiguity.

A required property for each submission of problem/outcome pairs is *reproducibility*: an unreproducible submission is useless and a waste of user bandwidth. When submissions of problem/outcome pairs are received they have to be validated for reproducibility. This can be achieved by keeping (possibly stripped down) copies of commonly used tools on the server side and by running them on the received problem description to check that the outcome matches the reported one. Given that we are not taking into account runtime upgrade errors, an error should manifest itself on the server side if and only if it has manifested itself on the user machine.

Together, the information items supported for submissions of problem/outcome pairs denote an outline called *DUDF skeleton*. In the skeleton, the following information items are *holes*:

(installer and meta-installer) package status, package universe, requested action, desiderata, outcome, and an optional use comment. Fully determined DUDF instances are made of this specification, together with distribution-specific documents describing how those holes are filled. A sketch of the DUDF skeleton is reported in Figure 2.1.

Holes are denoted by framed text. Additional information (annotations or *attributes*) of information items are reported in parentheses. The names used for information items are for presentational purposes, yet actually normative (see Section 2.4).

Note that in the skeleton, the package universe is sketched in its full generality: it is made of several package lists, each of which is annotated with its package list format. It is possible, though not granted, that to each package list corresponds a single file on the filesystem; in that case it is possible to annotate package lists with a *filename* containing the absolute paths corresponding to them. Also, it is possible that there exists an URL identifying the source from which the package list originated (e.g., where it was downloaded from on the web); in that case, it is possible to annotate package lists with a *url* representing the list source.

2.3 Extensional vs intensional sections

We have to minimize space consumption (in terms of bytes) in order not to discourage submissions by wasting the user's resources. In general, all the information items required for submissions are locally available on the user machine; in principle they are all to be sent as part of a submission. However, while some of the information items are *only* available on the user machine (e.g. current local package status and requested action) some other items can be grouped into parts stored elsewhere (e.g. package lists forming the current package universe) which have possibly been replicated on the user machine in a local cache.

We distinguish two alternative ways of sending submission information items (or sections): a section can either be sent intentionally or extensionally. An *extensional section* is a self-contained encoding of some information available on the user machine, for example a dump of the current local package database, or a dump of the current package universe.

An *intentional section* is a non self-contained encoding of some information available on the user machine, consisting of a *reference* pointing to some external resource. De-referencing the pointer, i.e. substituting the contents of the external resource for it, leads to the corresponding extensional section. For instance, several distributions have package repositories available on the Internet which are regularly updated. The current package universe for a given user machine may correspond to package indexes downloaded from one or several such repositories. A set of checksums of such indexes is an example of an intensional package universe section. Provided that a historical mirror of the distribution repositories is available somewhere, a corresponding extensional package universe can be built by looking up and then expanding the checksums in the historical mirror.

The use of intensional sections instead of extensional ones is the most straightforward space optimization we recommend to implement in collecting problem submissions. Here are some use cases for similar optimization:

- Most likely intentionality has to be used for the current package universe, though it will require setting up historical mirrors (the package metadata is sufficient for that, it will not be necessary to mirror the packages themselves).

- Even though the local package status appears to be a section that should forcibly be sent extensionally (as the information are not stored elsewhere), some partial intension can be designed for it.

  For example, assuming that the pair $\langle pkg\_name, pkg\_version \rangle$ is a key univocally determining a given package (*version uniqueness assumption*[1]), one can imagine sending as the local package status a set of entries $\langle\langle pkg\_name, pkg\_version \rangle, pkg\_status \rangle$, letting the server expand further package metadata (e.g. dependency information) on reception of the submission. In those rare cases where the version uniqueness assumption is not verified, the check for reproducibility is sufficient to spot non-reproducible submissions and discard them.

- The upgrade problem outcome has to be sent extensionally as to check for its reproducibility upon reception. Of course, the same optimizations as proposed in the previous point are applicable to outcomes in case of success.

Any section of a submission can be sent intentionally or extensionally, independently from the other sections; different choices can be applied to different submissions. In fact, the choices of how to submit the various sections are driven by the need of fulfilling the reproducibility requirement. For instance, if a given package universe is composed like the union of several remote package repositories, we will need to know all the involved packages, potentially coming from any repository in order to reproduce a submission. While a suitable intention might be available for some repositories, this may not be the case for some others (e.g. we might be lacking the needed historical mirror). In such a situation the proper solution is to send some repository reference intentionally, and the whole package listing of others extensionally.

It is up to the DUDF submission tool to know which parts of the package universe can be sent intentionally and which cannot.

2.4 Serialization

In this section we describe how to serialize any given instance of DUDF to a stream of bytes so that it can be serialized on disk (e.g. to create a local archive of problem descriptions to be submitted as a single batch) or over the network (for the actual submission to a distribution-specific problem repository).

The serialization of DUDF is achieved by describing a mapping from the DUDF skeleton to an XML [BPSM+06] tree. The actual serialization to bytes can then be done following the usual XML serialization rules.

To obtain the XML tree of a DUDF problem/outcome submission, one only needs to start from the corresponding outline (see Figure 2.1) and do the following:

1. Create a root element node called `dudf`, put it in the (default) namespace identified by `http://www.mancoosi.org/2008/cudf/dudf`.

2. Add an attribute `dudf:version`[2] to the root node, the value of which value is the value of the subsection `version` of the `dudf` section in the DUDF outline.

[1] This is assumption is not necessarily well-founded: users can rebuild packages locally, obtaining different dependency information, while retaining $\langle pkg\_name, pkg\_version \rangle$

[2] The namespace prefix `dudf:` is bound to `http://www.mancoosi.org/2008/cudf/dudf`

3. Starting from the DUDF outline root (and excluding the already processed `version` section), traverse the outline tree, adding child elements the general identifier of which is the section name used in the DUDF outline and the content of which is the result of recursively processing its content in the DUDF outline.

4. For annotated outline elements (e.g. package lists composing the package universe, which are annotated with format identifiers), map annotations to XML attributes of the relevant XML elements (note that the attributes should be explicitly prefixed with `dudf:`, as in XML attributes do *not* inherit the default namespace).

The same procedure is applied to obtain the XML tree of a DUDF sole problem submission, except that the outcome section (which should be missing anyhow in the starting DUDF outline) has to be skipped.

A non-normative example of serialization from the DUDF skeleton of Figure 2.1 to XML can be found in Appendix A, Figure A.1.

Chapter 3

Conclusion

The Mancoosi project will run a solver competition [DC08], in which each participant will try to find the best possible solutions to upgrade problems as those faced by users of FOSS software distributions. This document outlines the infrastructure for submitting problem instances from user machines to a distribution-independent repository.

The first class of document formats that are part of this infrastructure is DUDF (Distribution Upgrade Description Format), described in Chapter 2. Specific instances of DUDF will be used as document formats to encode real life problems encountered by users of FOSS software distributions. DUDF is meant to be a compact representation of upgrade problems, suitable to be transferred over the network. In addition to the purposes of the competition, DUDF documents might be useful to store and transfer the state of package managers, for example for reporting bugs concerning package managment tools.

Distributions that are interested in providing problems on which the competition will possibly be run should have an interest in implementing DUDF for their own distributions. The current document only describes the outline (or skeleton) of DUDF. Implementing DUDF actually means standardizing a specific instance of it, by describing in a separate document how the holes left open by this specification have to be filled in the context of a specific software distribution. Equipped with this specification and the specification of a DUDF instance, implementors will be able to produce and interpret DUDF corresponding to upgrade problems faced on final user machines.

The second important document format is CUDF (Common Upgrade Description Format). Its purpose is to provide a model in which upgrade problems can be encoded, by abstracting over distribution-specific details. In the context of the competition, the interest of CUDF is to encode problems on which the actual competition will be run. This way, participating solvers will not need to implement distribution-specific semantics, and will only have to reason about a self-constained problem. The CUDF format has first been described in Deliverable D5.1 [TZ08]; an up-to-date description can be found in [TZ09].

Appendix A

DUDF skeleton serialization example

This non-normative section contains an example of DUDF serialization to XML. The example is given in Figure A.1, which is the serialization of the DUDF skeleton given in Figure A.1. In the example, XML comments have been put in place of outline holes and other missing information.

```
<dudf version="1.0"
    xmlns="http://www.mancoosi.org/2008/cudf/dudf"
    xmlns:dudf="http://www.mancoosi.org/2008/cudf/dudf">
  <timestamp><!-- timestamp in RFC822 format --></timestamp>
  <uid><!-- unique problem identifier --></uid>
  <distribution><!-- distribution identifier --></distribution>
  <installer>
    <name><!-- installer name --></name>
    <version><!-- installer version --></version>
  </installer>
  <meta-installer>
    <name><!-- meta-installer name --></name>
    <version><!-- meta-installer version --></version>
  </meta-installer>
  <problem>
    <package-status>
      <installer><!-- installer status --></installer>
      <meta-installer>
        <!-- meta-installer status -->
      </meta-installer>
    </package-status>
    <package-universe>
      <package-list
          dudf:format=<!-- package list format identifier -->
          dudf:filename=<!-- package list absolute path --> >
        <!-- package list -->
      </package-list>
      <!-- ... other package lists ... -->
      <package-list
          dudf:format=<!-- package list format identifier -->
          dudf:filename=<!-- package list absolute path --> >
        <!-- package list -->
      </package-list>
    </package-universe>
    <action><!-- requested meta-installer action --></action>
    <desiderata><!-- meta-installer desiderata --></desiderata>
  </problem>
  <outcome dudf:result=<!-- one of: "success", "failure"--> >
    <error><!-- error description (result: "failure")--></error>
    <package-status> <!-- result: "success" -->
      <installer><!-- new installer status --></installer>
      <meta-installer>
        <!-- new meta-installer status -->
      </meta-installer>
    </package-status>
  </outcome>
  <comment>
    <!-- additional, user-provided information -->
  </comment>
</dudf>
```

Figure A.1: XML serialization skeleton of a DUDF problem/outcome submission

Appendix B

RELAX NG schema for DUDF

This non-normative section contains a RELAX NG [CM01] schema which can be used to check whether a given XML document represents a valid DUDF skeleton serialization. The schema only ensures that the skeleton part of the XML document is valid with respect to this specification, since the details about how holes are filled are distribution-specific.

Additional comments in the schema denote "side conditions"—e.g. the fact that dates should be in RFC882 format—which are not expressed by the schema itself, and which should be checked to ensure proper implementation of DUDF.

The RELAX NG schema is reported in Figure B.1.

```
default namespace dudf = "http://www.mancoosi.org/2008/cudf/dudf"

any = ( element * { any* } | attribute * { text }* | text )

tool_id = (
   element name { text },        # must be a package name
   element version { text }      # must be a version number
)

package_status =
   element package-status {
      element installer { any* },        # installer-specific
      element meta-installer { any* }?  # meta-installer-specific
   }

start = element dudf {
   attribute dudf:version { "2.0" },
   element timestamp { text },  # must be a date in RFC822 format
   element uid { text },
   element distribution { text },
   element installer { tool_id },
   element meta-installer { tool_id },
   element problem {
      package_status,
      element package-universe {
         element package-list {
            attribute dudf:format { text }?,
            attribute dudf:filename { text }?,  # must be an
                                                # absolute path
            attribute dudf:url { xsd:anyURI }?,
            any*
         }+
      },
      element action { text },
      element desiderata { text }?
   },
   (element outcome {
      attribute dudf:result { "success" },
      package_status
   }
    | element outcome {
       attribute dudf:result { "failure" },
       element error { text }
    }),
   element comment { any* }?
}
```

Figure B.1: RELAX NG schema for the DUDF skeleton

Appendix C

Changes from previous versions

C.1 From deliverable D5.1 to version 2.0

- DUDF skeleton, Chapter 2:
  - add support for an optional *url* attribute on package lists, pointing to the package list origin
  - add support for additional, user-provided information, to be shipped via the new `comment` element
- Appendix B: fix the RELAX NG schema reported in so that the `anyURI` data type used on the `url` attribute is qualified.

Bibliography

[BPSM$^+$06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. `http://www.w3.org/TR/REC-xml`, August 2006. W3C Recommendation.

[Bra97] S. Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119 (Best Current Practice), March 1997.

[CM01] James Clark and Makoto Murata. RELAX NG specification. OASIS specification, 2001.

[Cro82] D. Crocker. Standard for the format of ARPA Internet text messages. RFC 822 (Standard), August 1982.

[DC08] Roberto Di Cosmo and Sophie Cousin. Project presentation. Deliverable D1.1, The Mancoosi project, January 2008. `http://www.mancoosi.org/deliverables/d1.1.pdf`.

[EDO06] EDOS Work Package 2 team. Report on formal management of software dependencies. Deliverable WP2-D2.2, The EDOS project, March 2006.

[TZ08] Ralf Treinen and Stefano Zacchiroli. Description of the CUDF format. Deliverable D5.1, The Mancoosi project, November 2008. `http://www.mancoosi.org/deliverables/d5.1.pdf`.

[TZ09] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report TR003, The Mancoosi project, November 2009. `http://www.mancoosi.org/reports/tr3.pdf`.