Common Upgradeability Description Format (CUDF) 2.0

Ralf Treinen
Stefano Zacchiroli

Technical Report 003

Version 2.0

24 November 2009

Abstract

The solver competition which will be organized by Mancoosi relies on the standardized format for describing package upgrade scenarios. This document describes the *Common Upgradeability Description Format* (CUDF), the document format used to encode upgrade scenarios, abstracting over distribution-specific details. Solvers taking part in the competition will be fed with input in CUDF format.

The format is not specific to Mancoosi and is meant to be generally useful to describe upgrade scenarios when abstraction over distribution-specific details is desired.

Status of this Document

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

Note: to highlight the difference between the previous format version (commonly referred to as "CUDF 1.0"), this document start as version 2.0 and can be referred to as "CUDF 2.0".

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

1 Introduction **5**

1.1 Two different upgrade description formats 5

1.2 CUDF Design Principles 6

2 Common Upgradeability Description Format (CUDF) **9**

2.1 Overview of CUDF 9

2.2 Content 11

2.2.1 Generalities 12

2.2.2 Types 13

CUDF type library 14

2.2.3 Core property schemata 22

Preamble 22

Package description 23

Request description 25

2.2.4 Document structure 26

Global constraints 26

2.3 Formal semantics 27

2.3.1 Abstract syntax and semantic domains 28

2.3.2 Installations 29

2.3.3 Consistent package descriptions 29

2.3.4 Semantics of requests 31

2.3.5 Comments on the semantics 32

2.4 Serialization 33

2.4.1 Overall CUDF file syntax 33

2.4.2 Information item serialization 33

Line continuations 34

2.4.3 CUDF file parsing 35

2.5 Generating CUDF 36

3 Conclusion **38**

A CUDF serialization example **40**

B Solver output format **43**

C Changes from previous versions **45**

C.1 From CUDF 1.0 to CUDF 2.0 45

List of Figures

2.1 CUDF overall structure 12
2.2 CUDF types and their relationships. 15
2.3 CUDF detailed structure 27

Chapter 1

Introduction

The objective 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 descriptions of upgrade scenarios (or "problems") 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 specifies of document format used to represent problem instances in UPDBs.

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 the context of a particular distribution. On the other hand, problem descriptions as used for

[1] Throughout this specification, the word "*problem*"—as in "upgrade *problem*"—is used in the sense of "problem solving". Hence, an "upgrade problem" is 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 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. The DUDF format is decribed in a separate document [TZ09b].

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 The remainder of this chapter lays out the design principle behind he CUDF format. Chapter 2 contains the specification of both syntax and semantics of CUDF; that 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 CUDF.

1.2 CUDF Design Principles

The next chapter contains the specification of the Common Upgradeability Description Format (CUDF). The purpose of such a format is to encode real upgrade problems, abstracting over details specific to a software distribution or a package manager, so that problems coming from different environments can be compared and treated uniformly. For the specific purposes of Mancoosi, upgrade problems encoded in CUDF format will be used to form a corpus of problems to be used in a solver competition.

The specification of CUDF is guided by the following general design principles.

Be agnostic towards distribution details The main purpose of CUDF, as reflected by its name, is to be a *common* format to be used to encode upgrade problems coming from heterogeneous environments. The main environments we are considering are FOSS distributions, but other software deployment platforms face similar upgrade needs. As a consequence, the key design principle of CUDF is to be agnostic with respect to distribution specific details such as the used package system, the used installer and meta-installer, etc. The final goal is to be able to compare problems coming from different platforms in a uniform setting, including at the very minimum all platforms for which a DUDF implementation (see [TZ09b]) has been provided.

Stay close to the original problem There are several encodings that can be considered after removing all distribution-specific details [MBC+06]. Since CUDF aims to be as close as

possible to the original problem we choose to avoid an encoding where the characteristic features of the original problem are abstracted away and are no longer distinguishable. There are various reasons backing this choice:

- Preserve the ability for humans to inspect problems encoded in CUDF.

  Ideally, users having submitted a problem (via submission of a DUDF document) should be able to look at their CUDF encoding and recognize the upgrade situation.

- Avoid bias towards specific upgrade planning techniques and implementations.

  Specific encodings (e.g. using a representation in propositional logic, or geared to constraint programming) bear the risk of giving an advantage or disadvantage to certain resolution techniques. Since one of our goals is to provide a set of problems to stimulate the advancement in upgrade planning, CUDF strives to stay independent of specific techniques and implementation details.

- Make life easy to legacy tools (installers and meta-installers) to interact with CUDF.

  Ideally, we want legacy tools to be able to take part in the solver competition we are organizing. That would be easy to achieve as long as the CUDF encoding still resembles something with which installers and meta-installers are familiar. Conversely, using an encoding that is too abstract would constitute an obstacle for the state-of-the-art tools.

Extensibility CUDF has no explicit support for specifying optimization criteria that whould allow to select the "best" possible solution among all possible solutions of a given upgrade problem. Optimization criteria can be expressed in a separate language [TZ09a]. Criteria expressed in that language will make use of additional data fields of package metadata that are only relevant for determining the optimality of a solution, but which are not relevant for CUDF itself. This leads to the need of having an *extensible* format to encode upgrade problems and in particular package metadata. Indeed, since we cannot anticipate all possible interesting optimization criteria we can neither anticipate all the metadata that shall be stored in CUDF documents. Hence the CUDF specification establishes a type system to be used for typing package metadata (see Section 2.2.2) and a set of core set of package metadata (see Section 2.2.3). Additional metadata can be added in the future by providing their schemata, in terms of the available types.

For example, to run a competition track in which the installed size of all packages on the system should be minimized, the track organizers can state that, in addition to the core package metadata, each package must be equipped with an `Installed-Size` property, the type of which is `posint`. The track rules will then describe how to determine the best solution, on top of the semantics of positive integers.

Transactional semantics Problems are encoded in CUDF according to the *point of view of the meta-installer*, rather than to the point of view of an installer. This means that our notion of solution correctness (see Section 2.3) considers the resulting package status and not *how* that status is obtained on the target machine. In particular, the order of package installations and removals or even the various phases of package deployment and installation (downloading, unpacking, etc.) are beyond the scope of the CUDF encoding.

In a sense, CUDF assumes that it is possible to pass from the package status as described in a CUDF document to any (correct) status found by a meta-installer in a *transactional* way. As an

example of a practical implication of this design principle, CUDF does not distinguish between Debian's `Depends` and `Pre-Depends`; note that this is coherent with the semantic encoding of [MBC+06], from which the CUDF semantics takes its inspiration.

Use plain text format On a more technical side, CUDF aims to be a *simple to parse* (read) and *simple to generate* (write) format. The reason is as simple as our interest in providing a tool to reason about future better upgrade planners, ignoring distracting details such as parsing or pretty printing. Plain text is the universal encoding for information interchange formats in the Free Software community [Ray03], using a plain text format makes it easy for contenders to adapt tools to our format. Moreover, it is an implicit need if we want users to be able to "look" at CUDF problems and understand them, without the need of specific tools. Similarly, this principle also implies that standard serialization formats should be preferred for CUDF. In fact, the CUDF specification describes the informative content of a CUDF document and its semantics on one hand, and how to serialize that content to disk (using already existing standards and technologies) on the other hand.

Chapter 2

Common Upgradeability Description Format (CUDF)

2.1 Overview of CUDF (non-normative)

This section gives an overview of the syntax and semantics of CUDF. A precise definition of the CUDF format will be given in Section 2.2, while a rigorous definition of its semantics will be given in Section 2.3. The current section is not normative, please refer to Sections 2.2 and 2.3 for precise definitions.

A CUDF document consists of a list of an optional preamble (which is not interesting for the purpose of this introduction), followed by several package description items, and finally a user request. In the concrete representation (see Section 2.4) each item is a stanza consisting of one or several (logic) lines of text. It is recommended that successive stanzas be separated by empty lines even though this is not mandatory.

Every line in a stanza starts with the word denoting the first property of that stanza, followed by the `":  "` separator and then the value of the property, an exception to this rule is the line "`request:`" which starts the stanza describing the query, and which does not necessarily have a meaningful value. Other properties of the same stanza come next, following the same serialization conventions. Properties with long values can be split on multiple physical lines using *line continuations*: when a line starts with an empty space it is considered to be a continuation of the property of the previous line; the actual property value is obtained by joining together the first property line together with all its continuations (removing intervening new lines and one heading space per continuation).

A package description stanza starts with the property `package` the value of which is the name of the package. Package names are non-empty strings, starting with a lowercase ASCII letter, and containing only lower or uppercase Latin letters (case is significant), numerals, dashes `"-"`, pluses `"+"`, dots `"."`, slashes `"/"`, and percents `"%"`. The order of all other properties in a package description stanza is not specified.

The only other mandatory property, besides `package`, in a package description stanza is `version`, the value of which is a positive (non-null) integer value. There may be at most one package description stanza for any given pair of package name and version. However, it is allowed to have in the same document different versions of the same package.

Then there are a number of properties that are relevant for the formal semantics but that are

only optional since there are default values that are used when the value is not explicitely set:

- The `installed` property (the values of which are of type `bool`, with default value `false`) indicates whether a package is installed or not. It is a priori allowed to have several versions of the same package installed. The setting of this field in the stanzas of a CUDF document describes the "initial" configuration of a machine, i.e. the configuration in which the user request is evaluated.
- The `keep` property has as possible values `version`, `package`, `feature`, or `none`. This value is only relevant in case the `installed` property is `true`. Package installations may evolve by changing the `installed` property associated to pairs of package name and package version. The `keep` property constraints the possible evolutions of the installation:

  `version` means that this particular version of a package must not be removed,

  `package` means that at least one version of that package must remained installed,

  `feature` means that all features (see below) provided by this version of the package must continue to be provided,

  `none` puts no constraint on possible evolutions of the installation.

  The default value of that property is `none`.

Then there are three properties which define relations between packages:

- The `provides` property is a possibly empty list of names of so-called features, also called *virtual packages*. In this list, features may be declared either by giving an exact version, or without mentioning a version. Features are frequently used in RPM-like packaging system to declare the fact that a package installs a particular file on disk, and also both in RPM and Debian-like packaging systems to declare that a package provides a certain abstract functionality, like for instance *mail-transport-agent* or *postscript-reader*. A list of several features is interpreted as that package realizing all the features in the list, with the version as given in the list, or of *all possible versions* when no particular version is mentioned in the list.

  The default value of that property is the empty list (that is, no feature is provided).

- The `depends` property has as value a complex dependency on the existence of packages or on features. Simple dependencies are given as the name of the package or feature, and may carry in addition a constraint on the version number. Version constraints can be of any of the form `=` v, `!=` v, `>` v, `<` v, `<=` v or `>=` v where v is a version number. Complex dependencies are obtained by combining dependencies with conjunctions (denoted by `","`) and disjunctions (denoted `"|"`). However, dependencies are limited to so-called conjunctive normal forms, that is conjunctions of disjunctions.

  The default value of this property is the formula `True` (that is no particular dependency constraint).

- The `conflicts` property has as value a list of packages (or features), possibly equipped with package-specific version constraints. Version constraints are the same as for the `depends` property. Such a conflict list describes a list of packages that *must not* be installed. For instance, if package p of version 5 conflicts with package `q >= 7` then we are not allowed to install version 5 of p together with any of the versions 7 or greater of q. However, it would be allowed to install version 5 of p together with version 6 of q.

There is a special treatment for so-called *self-conflicts*: any conflicts stemming from a pair of package p and version v are ignored when checking the conflicts of this pair (p, v). For instance, when package p of version 5 indicates that it conflicts with package p (without version constraint) this means that version 5 of package p cannot be installed together with any other version of p. A conflict of package p in version 5 with package p in version 5 is allowed as a special case but does not have any effect.

Self-conflicts of this kind are often used by packaging systems in order to express that only one (version of a) package implementing a certain feature may be installed at any given time. For instance, both the package *sendmail* and the package *postfix* (of any version) may provide the feature *mail-transport-agent* and also conflict with *mail-transport-agent*. The effect of this is that it is not possible to install *sendmail* and *postfix* together (or any of them together with any other package providing *mail-transport-agent*), but it does allow to install *sendmail* or *postfix* since the conflict stemming from the package itself is ignored.

The default value of this property is the empty list (that is, no conflict declared here).

Finally, the CUDF document contains a stanza representing the user request. This stanza starts with the line `request:`, and it may contain an `install` property, a `remove` property, and a `upgrade` property. All these properties are optional, their value is a list of packages (or features) possibly equipped with version constraints; the default value of these three properties is the empty list. The `install` property gives packages that are requested to be installed, while the `remove` gives packages that must be removed. The `upgrade` property has a similar meaning as `install`, the difference being that the former requires that for every package in that list only one version be installed, and that this version be greater or equal to any version of that package previously installed.

The request stanza is headed by a few extra (and mostly optional) information such as a unique identifier of the CUDF document at hand and a set of checksums encompassing different parts of the document, so that solvers processing it can cache relevant derived information.

2.2 Content

A CUDF document (or simply "CUDF") is composed of a set of *information items*. Each item represents a part of the original upgrade problem.

Each information item belongs to exactly one of the following classes:

Preamble specify global (meta-)information about the CUDF document that contains it. Generally, such information are needed or simply useful to better process the containing document.

The preamble information item is optional. A CUDF document must contain at most one preamble information item.

Package description specify packages that play a role in the upgrade problem described by a given CUDF.

A CUDF document contains several package descriptions. In a typical scenario there is one such item for each package known to the package manager, including both locally installed packages (as part of the local status) and packages available from remote repositories known to the meta-installer (as part of the package universe).

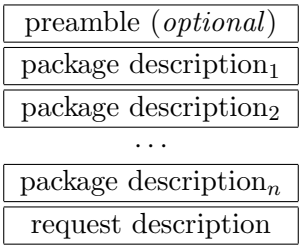

Figure 2.1: Overall structure of a CUDF document.

Request description describe the user upgrade request that has been submitted to the package manager. Such information is not specific to any particular package, in that sense it is global and hence does not belong to any package-specific information item.

A CUDF document must contain exactly one request description item.

CUDF documents consist of an optional preamble item, followed by several package description items, followed in turn by a single request description item.[1]

The order is fixed: preamble (if any) comes first, package descriptions come next, request description comes at the end. A schematic representation of a typical CUDF is given in Figure 2.1. A similar, yet more detailed, pictorial representation of CUDFs is given later on in Figure 2.3.

2.2.1 Generalities

Each information item consists of a set of *properties*. Each property has a name and a value, i.e., each property consists of a $\langle \text{name}, \text{value} \rangle$ pair. Any given information item is then a set of pairs $\{\langle n_1, v_1 \rangle, \ldots, \langle n_k, v_k \rangle\}$. In such a set, property names are unique, that is $\{n_1, \ldots, n_k\}$ contains no duplicate, i.e., property names can be used as keys to look up property values.

A *property name* is a string of Unicode [Con06] characters matching the additional lexical constraint of being an *identifier*; the actual constraints are given in the definition of the `ident` type (see Section 2.2.2 below).

A *property value* is a typed value, belonging to some set. We call this set the *set of values* or the *type domain* of the type associated to a property. The type is fixed for each property: any given property can only assume values having the very same type and coming from the very same set of values; the description of each supported property in this specification states what is the type of its values.

Optional vs required properties A property can be either optional or required. A property is *optional* if its value is marked as such in the property description, otherwise it is *required*. Required properties must always be present as part of the information items they belong to, while optional properties may be omitted.

Optional properties that are not present in CUDFs are treated as properties assuming their default values. It is indistinguishable whether the value was actually specified in the CUDF serialization or not.

[1] There is no constraint on the number of package description items, but problems represented as CUDFs with no package description items are uninteresting. CUDFs are expected to include at least one package description item, and usually many more than just one.

Property schemata Each property supported by CUDF can be fully specified using a *property schema*. Such a schema consists of:

- the name of the property;
- the type of property values;
- the information item the property belongs to;
- the optionality of the property (i.e. whether the described property is required in each instances of the information item it belongs to), optionality is either "required" or "optional";
- for optional properties only, their default value. It is mandatory to specify a default value for optional properties.

Actual CUDF documents must contain all required properties for each information item. For both required and optional properties, the type of property values must match the type prescribed by property schemata.

Extra (package) properties Section 2.2.3 gives the schemata of the *core property set* supported by CUDF. Nevertheless the set of properties which can be given to build information items is open-ended (the so called *open-world assumption*), and not restricted to the core set. Information items can contain *extra properties* not prescribed by this specification as long as their names do not clash with names of properties in the core property set.

It is up to implementations to make use of such extra properties. However, all extra properties must be declared in the CUDF preamble using the `property` (meta-)property (see Section 2.2.3). Extra properties for information items other than package descriptions are currently not supported and, as a consequence, must not appear within preamble and request description information items.

2.2.2 Types

As discussed above, each property value has a type which is fixed for any given property. A *type* is a set of values, which is also called *value space* or *domain* of a given type. Let t be a type, we denote with $\mathcal{V}(t)$ its value space. Moreover, a *lexical space* $\mathcal{L}(t)$ is associated to each type, and it denotes the set of possible representations of all values belonging to the value space as strings of Unicode characters. The relationships between the value spaces and lexical spaces are as follows:

- For each $l \in \mathcal{L}(t)$ there is a unique corresponding value $parse_t(l) \in \mathcal{V}(t)$. The function $parse_t(\cdot)\colon \mathcal{L}(t) \to \mathcal{V}(t)$ is the *parsing* function used to parse syntactic values into semantic values.
- For each $v \in \mathcal{V}(t)$ there can be several $l \in \mathcal{L}(t)$ such that $v = parse_t(l)$, i.e. the parsing function is not necessarily one-to-one.

Subtyping Interesting relationships also exist between types, in particular *subtyping*. A type t_2 is said to be a subtype of a *supertype* t_1 (written $t_2{<:}t_1$) if $\mathcal{V}(t_2) \subseteq \mathcal{V}(t_1)$, that is, when the domain of the subtype is contained in the domain of its supertype. Given $t_2{<:}t_1$, the lexical space of t_2 can be obtained by restricting the lexical space of t_1 to all elements which can be parsed to elements of the value space of t_2, i.e. $\mathcal{L}(t_2) = \{l \in \mathcal{L}(t_1) \mid parse_{t_1}(l) \in \mathcal{V}(t_2)\}$. Therefore the parsing function for a given subtype can be obtained by simply reusing the parsing function of the supertype treating as parsing errors all values not belonging to the domain of the subtype.

As a consequence of the above definitions and properties, each type can be defined by describing its value and lexical spaces, as well as the semantics of its parsing functions. Subtypes can be defined by simply giving restrictions on the value space of supertypes. The section further gives the definitions for all types used by CUDF.

Conventions In this specification abstract values belonging to the value space are denoted using mathematical notation.

Lexical values are denoted by double-quoted strings typeset in `monospace font` and encoded in UTF-8. The double-quotes are used for presentational purposes of this specification and are not part of the actual lexical value. Such a value can be found by considering the Unicode string corresponding to the given UTF-8 string, after having removed double quotes. For example, the lexical value `"foo"` denotes the Unicode string of length 3, composed of the three lowercase letters "f" (Unicode code point `U+0066`), "o" (`U+006F`), and "o" again.

Functions can either be described intentionally or extensionally. In the former case, types are specified via natural language explanation of their semantics, or reference to functions described elsewhere. In the latter case they are defined by enumerating argument/result pairs using the following notation: $\{input_1 \rightarrow output_1, \ldots, input_n \rightarrow output_n\}$.

For the sake of brevity, several details about lexical values and parsing functions are deferred to external specifications, most notably to "XML Schema Part 2: Datatypes" [BM04], which specify a set of simple datatypes, providing for each of them notions similar to the one introduced above: value space, lexical space and parsing functions. When deferring a definition to the definition of the corresponding XML Schema datatype, we will write $\texttt{xs:foo}^{\text{XML Schema}}$, where "xs:foo" is the XML Schema datatype name.

Complex lexical spaces are sometimes described by the means of EBNF grammars [EBN96] which use SmallCaps for non terminal symbols and double-quoted strings as described above for terminals. Grammars are always given with the productions of their start symbol first. In order to avoid duplications, grammars appearing later on in this specification can reuse symbols defined in previous grammars. When EBNF grammars are used, the definition of parsing functions can be omitted and delegated to parsers built using the given grammar. For the completeness of this specification it is enough to state how the values associated to non terminals have to be translated to elements in the value space (i.e. to give the "semantic actions" associated to grammar productions).

CUDF type library

In the presentation of the available CUDF types that follows, we first introduce all *primitive types*, i.e. all those types that are not obtained via subtyping; then we describe *derived types*, i.e. those that are obtained as subtypes of other (primitive or derived) types. As discussed

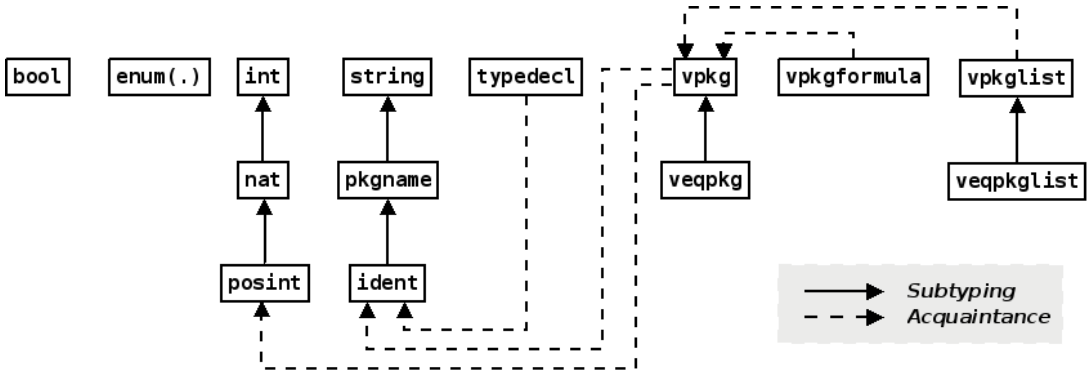

Figure 2.2: CUDF types and their relationships.

above, each derived type can be described by simply giving a restriction of the value space of its supertype.

For the lexical definition of some types, we will make use of the following convenience classes of Unicode characters:

Lowercase Latin letters from "a" (Unicode code point U+0061) to "z" (U+007A).

Uppercase Latin letters from "A" (Unicode code point U+0041) to "Z" (U+005A).

Latin letters the union of lowercase and uppercase Latin letters.

Arabic numeral digits from "0" (U+0030) to "9" (U+0039).

Figure 2.2 shows a diagram giving an overview of CUDF types and their relationships. In the figure, directed straight arrows denote subtyping relationships, with the type pointed at being the supertype; directed dashed arrows denote acquaintance, i.e. the fact that the values of one type contain values of other types, the latter being pointed at by the arrow. Transitive relationships are omitted from the figure.

NAME	`bool`
DESCRIPTION	Boolean values
VALUE SPACE	The set of distinguished values {`true`, `false`}
LEXICAL SPACE	The set of strings { `"true"`, `"false"`, }
PARSING	{`"true"` → `true`, `"false"` → `false`}

NAME	`int`
DESCRIPTION	Integer numbers
VALUE SPACE	The set of integer numbers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (Note that this set is infinite, unlike legacy integers available in most programming languages.)
LEXICAL SPACE	The same lexical representation as the one used by the `xs:integer`$^{\text{XML Schema}}$, i.e. finite-length sequences of decimal digits (U+0030–U+0039) with an optional leading sign (defaulting to `"+"`). For example: `"-1"`, `"0"`, `"12678967543233"`, `"+100000"`.
PARSING	The same parsing function as the one used for `xs:integer`$^{\text{XML Schema}}$

NAME	`string`
DESCRIPTION	Unicode strings
VALUE SPACE	The set of—possibly empty—single-line strings of arbitrary Unicode characters. With "single-line" we mean all Unicode characters except line feed (`U+000A`) and carriage return (`U+000D`).
LEXICAL SPACE	Some (specified) character encoding supported by Unicode. For the purpose of CUDF serialization the character encoding of choice is UTF-8 (see Section 2.4).
PARSING	The same parsing functions as the ones used for $\texttt{xs}:\texttt{string}^{\text{XML Schema}}$, i.e. simply decoding from the used character encoding to Unicode character strings.

We assume the notion of the function *length* for Unicode strings, which is defined by counting the number of Unicode characters (not bytes) forming a Unicode string in a given encoding. As a consequence, the empty string `""` has length 0.

NAME	`vpkg`
DESCRIPTION	Versioned package names
VALUE SPACE	The set of pairs $\langle vpred, p\rangle$ where p is a value of type `pkgname` (see below) and $vpred$ is either $\top$ (denoting that no version constraint has been imposed on package name p) or a pair $\langle relop, v\rangle$ (denoting that a specific version constraint is in effect on package name p). In the latter case $relop$ is one of $\{=, \neq, \geq, >, \leq, <\}$ and v is a value of type `posint` (see below).
LEXICAL SPACE	The set of strings matching the grammar:

VPKG ::= PKGNAME (SP + VCONSTR)?
VCONSTR ::= RELOP SP + VER
RELOP ::= `"="` | `"!="` | `">="` | `">"` | `"<="` | `"<"`
SP ::= `U+0020` (i.e. space) | `U+0009` (i.e. tab)
VER ::= POSINT

where the nonterminal PKGNAME matches lexical values of `pkgname` (see below) and POSINT those of `posint` (see below). The values resulting from parsing VCONSTR, which match RELOP and VERSION respectively, are used to form the internal pair $\langle relop, v\rangle$; similarly, the values resulting from parsing VPKG are used to form the external pair $\langle vpred, p\rangle$.

PARSING	Induced by the grammar. RELOP is parsed by the function: $\{\texttt{"="} \to =, \texttt{"!="} \to \neq, \texttt{">="} \to \geq, \texttt{">"} \to >, \texttt{"<="} \to \leq, \texttt{"<"} \to <\}$.

The semantics of versioned package names depend on the context where they appear. Generally, package names without version constraints are to be intended as package predicates matching all packages with a given name. Package names with a version constraint match only packages with the given name and a version satisfying the version constraint.

NAME	`vpkgformula`
DESCRIPTION	Formulae over versioned package names
VALUE SPACE	The smallest set F such that:

$$\begin{array}{lll}
\texttt{true} \in F & & \text{(truth)} \\
\texttt{false} \in F & & \text{(untruth)} \\
\mathcal{V}(\texttt{vpkg}) \subseteq F & & \text{(package predicate)} \\
\bigvee_{i=1,\ldots,n} a_i \in F & a_1, \ldots, a_n \text{ atoms} \in F & \text{(disjunctions)} \\
\bigwedge_{i=i,\ldots,n} d_i \in F & d_1, \ldots, d_n \text{ disjunctions} \in F & \text{(conjunctions)}
\end{array}$$

LEXICAL SPACE The set of strings matching the following grammar:

$$\begin{array}{rcl}
\textsc{Fla} & ::= & \textsc{AndFla} \mid \texttt{"true!"} \mid \texttt{"false!"} \\
\textsc{AndFla} & ::= & \textsc{OrFla}\ (\textsc{Sp}*\ \texttt{","}\ \textsc{Sp}*\ \textsc{OrFla})* \\
\textsc{OrFla} & ::= & \textsc{AtomFla}\ (\textsc{Sp}*\ \texttt{"|"}\ \textsc{Sp}*\ \textsc{AtomFla})* \\
\textsc{AtomFla} & ::= & \textsc{VPkg}
\end{array}$$

PARSING Induced by the grammar.
ATOMFLA nonterminals are parsed as atoms, ORFLA as disjunctions of the atoms returned by their ATOMFLAS, ANDFLA as conjunctions of the disjunctions returned by their ORFLAS. The terminals `"true!"` and `"false!"` represent `true` and `false` respectively.

Note that formulae over versioned package names are always in conjunctive normal form (CNF), i.e. they always have the shape of "conjunctions of disjunctions of atomic formulae".

NAME	`vpkglist`
DESCRIPTION	Lists of versioned package names
VALUE SPACE	The smallest set L such that:

$$\begin{array}{lll}
[] \in L & & \text{(empty lists)} \\
p::l \in L & p \in \mathcal{V}(\texttt{vpkg}), l \in L & \text{(package concatenations)}
\end{array}$$

LEXICAL SPACE The set of strings matching the grammar:

$$\textsc{VPkgs} \ ::= \ \texttt{""} \mid \textsc{VPkg}\ (\textsc{Sp}*\ \texttt{","}\ \textsc{Sp}*\ \textsc{VPkg})*$$

PARSING Induced by the grammar.
`""` is parsed as [], while an instance of VPKG followed by a list of versioned package names is parsed as package concatenation.

NAME	`enum`$[s_1, \ldots, s_n]$
DESCRIPTION	Enumerations
VALUE SPACE	Rather than a single type, `enum` is a type scheme defining infinitely many possible actual types. All these types are indexed by a set of symbols $\{s_1, \ldots, s_n\}$, for any such set a single type (an *enumeration*) is defined. Each enumeration is a type, its values can be one of the symbols $s_1, \ldots, s_n$. Symbols must match the lexical constraints of the `ident` type, and one must have $n \geq 1$. For convenience, in this specification symbols are written as strings, but without the external double quotes.
LEXICAL SPACE	$\{s \in \mathcal{L}($`string`$) \mid s$ belongs to the lexical space of `ident` (see below)$\}$
PARSING	$\{$`"`s`"` $\rightarrow s \mid s$ belongs to the lexical space of `ident` (see below)$\}$ The parsing function is defined point-wise on each Unicode string matching the lexical space of `ident`. For each of them, the parsing function returns a symbol, the name of which is that very same identifier.

For example, given the enumeration $E =$ `enum`$[$`foo`$,$`bar`$,$`baz`$]$, we have the following: `foo` $\in \mathcal{V}(E)$, `bar` $\in \mathcal{V}(E)$, and `baz` $\in \mathcal{V}(E)$.

NAME	`pkgname`
DESCRIPTION	Package names
VALUE SPACE	`pkgname` is a subtype of `string`. It is obtained allowing only strings that satisfy the following condition in the value space: • the string only consists of: Latin letters (lower or uppercase), Arabic numeral digits, dashes (`U+002D`), pluses (`U+002B`), dots (`U+002E`), slashes (`U+002F`), commercial at (`U+0040`), (round) parentheses (`U+0028` and `U+0029`), and percents (`U+0025`). • the string has length 1 or greater
LEXICAL SPACE	As per subtyping.
PARSING	As per subtyping.

Values belonging to $\mathcal{V}($`pkgname`$)$ are used to reference packages by name. Syntactically, package names are more liberal than identifiers and allow more characters. This is intended to help encoding (virtual and real) package names coming from different distributions.

For all characters that still do not belong to package names the percent sign can be interpreted as an escaping character followed by two (or more) hexadecimal digits as it happens with URI escaped encoding [BLFM98]. Note, however, that CUDF does no interpretation whatsoever of the escaping, e.g. a package called "`%61`" has precisely that name and is different from a package called "`a`".

NAME	`ident`
DESCRIPTION	Identifiers
VALUE SPACE	`ident` is a subtype of `pkgname`. It is obtained allowing only strings that satisfy the following condition in the value space: • the string starts with a lowercase Latin letter • the string only consists of: lowercase Latin letters, Arabic numeral digits, and dashes (`U+002D`). • the string has length 1 or greater
LEXICAL SPACE	As per subtyping.
PARSING	As per subtyping.

Identifiers are used to reference various named entities throughout the CUDF specification. In particular, they are used to reference property names and enumeration items.

NAME	`nat`
DESCRIPTION	Natural numbers
VALUE SPACE	`nat` is a subtype of `int`. It is obtained by allowing only non-negative integers in the value space.
LEXICAL SPACE	As per subtyping.
PARSING	As per subtyping.

NAME	`posint`
DESCRIPTION	Positive natural numbers
VALUE SPACE	`posint` is a subtype of `nat`. It is obtained by removing the number 0 from the value space of `nat`.
LEXICAL SPACE	As per subtyping.
PARSING	As per subtyping.

NAME	`veqpkg`
DESCRIPTION	Version-specific package names
VALUE SPACE	`veqpkg` is a subtype of `vpkg`. It is obtained by removing all packages with version constraints other than $=$, more formally: $\mathcal{V}(\texttt{veqpkg}) = \{\langle vpred, p\rangle \mid \langle vpred, p\rangle \in \mathcal{V}(\texttt{vpkg}), vpred = \top \vee vpred = \langle =, v\rangle \text{ for some } v\}$ from the value space of `vpkg`.
LEXICAL SPACE	As per subtyping.
PARSING	As per subtyping.

NAME	`veqpkglist`
DESCRIPTION	Lists of version-specific package names
VALUE SPACE	`veqpkglist` is a subtype of `vpkglist`. It is obtained by using as value space only the smallest set $L' \subseteq \mathcal{V}(\texttt{vpkglist})$ such that: $[] \in L'$ (empty lists) $p::l \in L' \quad p \in \mathcal{V}(\texttt{veqpkg}), l \in L'$ (package concatenations)
LEXICAL SPACE	As per subtyping.
PARSING	As per subtyping.

Finally, we introduce the special type used to declare type declaration relationships between identifiers (representing properties) and types. Even though it is not a subtype of any other type, we have postponed its declaration thus far because it is less relevant than the former types to understand the inner workings of CUDF.

NAME	`typedecl`
DESCRIPTION	Type declarations
VALUE SPACE	A single *type declaration* is a triple $\langle i, t, v \rangle$ where i is an identifier, t is a type (that is, one of the types declared in the CUDF type library) other than `typedecl`, and v is either $\top$ or a value of type t. The intuitive meaning of a single type declaration is as follows. The identifier i is declared to denote values of type t in some context (usually, a package description information item, see the description of the `property` property in Section 2.2.3); if $v = \top$, then i is declared to be required and has no default value, otherwise it is declared to be optional and has v as its default value. Let $D = \langle i, t, v \rangle$ be the set of all type declarations, then the value space of `typedecl` is obtained by taking the powerset $\mathcal{P}(D)$ (the set of all sets of type declarations) and removing from it all sets of type declarations such that identifiers i are not unique. Intuitively, a single value of `typedecl` is a set of property schemata. Using such a value, property schemata can be looked up by identifier to additionally find out: the matching type, optionality, and default value (if any).

LEXICAL SPACE The set of strings matching the grammar:

```
TypeDecls  ::=  "" | TypeDecl (Sp* "," Sp* TypeDecl)*
TypeDecl   ::=  Ident Sp* ":" Sp* TypeExpr
                (Sp* "=" Sp* "[" Value "]")?
TypeExpr   ::=  TypeName
             |  "enum" Sp* "[" Ident ("," Sp* Ident)* "]"
```

where the nonterminal IDENT matches lexical values of `ident`, TYPENAME matches the name of a type coming from the CUDF type library (with the exception of `enum`), and VALUE matches lexical values of `string`. The values coming from IDENT give the identifier of each type declaration triple, those coming from TYPEEXPR denote the type, the values coming from VALUE need special treatment. When TYPEEXPR is used to denote an `enum` type, its allowed enumerations are explicitly listed. If VALUE is missing then the third component of the type declaration is $\top$.

If VALUE is present, it needs to be parsed as a lexical value of the type denoted by TYPENAME. For instance, if the type is `int` it will be parsed by $parse_{\texttt{int}}$, it the type is `bool` it will be parsed by $parse_{\texttt{bool}}$ and so forth.

As a special case, if TYPENAME is `string`, then VALUE is an *escaped string* surrounded by double quotes characters (`U+0022`). To obtain the actual `string` value, such VALUE should be:

1. stripped of its surrounding double quotes
2. *unescaped* (see below)
3. parsed as usual using $parse_{\texttt{string}}$

Unescaping is defined as follows: all sequences of 2 characters having backslash (`U+005C`) as its first character and appearing on the left-hand side of the table below will be replaced with the corresponding single character on the right-hand side of the table:

escape sequence	*replace with*
\"	"
\\	\

PARSING Induced by the grammar.

The main purpose of type declarations in CUDF is to enable documents to declare in advance the type of package extra properties. In that specific case, the identifiers of type declarations are used as names of package extra properties.

For example, in such context the lexical value:

```
pin-priority: int , suite: enum[stable,testing,unstable] = [stable]
```

will be parsed as the type declaration declaring two extra package properties: a required property called `pin-priority`, having type `int`; and an optional property called `suite`, having an enumeration type (with allowed values: `stable`, `testing`, and `unstable`), and default value `stable`. For better readability, in actual CUDF documents the above lexical value will be usually typeset using line continuations (see Section 2.4.2).

```
property: pin-priority: int ,
 suite: enum[stable,testing,unstable] = [stable]
```

As an extreme example of string escaping, you can additionally define a property called `weird` of type `string` having default value of `""]"` (a string of length two composed by a double quote and a closing bracket) as follows:

```
property: pin-priority: int ,
 suite: enum[stable,testing,unstable] = [stable] ,
 weird: string = ["\"]"]
```

2.2.3 Core property schemata

Each of the information items supported by CUDF (see Section 2.2) is composed of several properties. In this section we give the schemata for all properties that can be part of the various kinds of description items.

Preamble

The optional preamble provides information useful for the processing of further parts of a CUDF document.

NAME	`property`
TYPE	`typedecl`
OPTIONALITY	optional
DEFAULT	{}
DESCRIPTION	Type declaration for extra properties used in package descriptions.

Using the `property` (meta-)property, the preamble is used to declare the types of extra package properties used in subsequent package descriptions. By default no extra properties are declared and, as a consequence, no non-core properties can be used in package descriptions.

NAME	`univ-checksum`
TYPE	`string`
OPTIONALITY	optional
DEFAULT	`""` (the empty string)
DESCRIPTION	Checksum taken over all properties of all package description stanzas, *with the exception* of the following properties: `installed`, `keep`. CUDF consumer applications are allowed to assume that, if two CUDF documents have the same `univ-checksum` and if that value is not the empty string, then they both have a set of packages which is equal; in particular the set of pairs ⟨package name, package version⟩ will be identical for the two documents (i.e. neither document is allowed to have a package that the other document is missing). Still, the two documents will be allowed to have packages with different values of properties excluded above (in particular, they will be allowed to have different sets of `installed=true` packages).

Ideally, identical values of the `univ-checksum` property will enable downstream solvers to cache derived information (e.g. clauses, available features, etc.) that depend on "stable" core package properties. The value of `installed` is not considered among those stable properties on purpose; not considering it allows solvers to cache information that is invariant under installation attempts, so that several installation attempts performed in a row (accumulating package status changes) can benefit from caching.

Note that the default value of the empty string does not allow to make any caching assumption. Also, note that what is granted to be equal for equal `univ-checksum` value is a *set* of packages, so the order in which packages appear in a given CUDF serialization is totally irrelevant.

NAME	`status-checksum`
TYPE	`string`
OPTIONALITY	optional
DEFAULT	`""` (the empty string)
DESCRIPTION	Analogous to `univ-checksum` but encompassing information that define package status, i.e. the set of installed packages. More precisely, `status-checksum` is defined over all packages and properties without excluding any specific property. In particular, that means that `status-checksum` also takes into account the following properties: `installed`, `keep`.

NAME	`req-checksum`
TYPE	`string`
OPTIONALITY	optional
DEFAULT	`""` (the empty string)
DESCRIPTION	Checksum taken over all properties of the request information item. Identical values of `req-checksum` on different CUDF documents would allow downstream solvers to assume that, the two documents express the same request (even though package statuses and/or package universes can differ in the two documents).

Note that the equality of both `status-checksum` and `req-checksum` on different CUDF documents would allow downstream solvers to return identical solutions.

Package description

A package description item describes several facets of a package.

NAME	`package`
TYPE	`pkgname`
OPTIONALITY	required
DESCRIPTION	Name of the package being described.

NAME	`version`
TYPE	`posint`
OPTIONALITY	required
DESCRIPTION	Version of the package being described.

Name	`depends`
Type	`vpkgformula`
Optionality	optional
Default	`true`
Description	Intentional representation of the dependencies of the package being described.

Dependencies indicate which packages need to be installed to make a given package installable. Dependencies are indicated as boolean CNF formulae over possibly versioned package names. Dependencies are the most expressive relationships which can be stated among packages using CUDF properties.

Name	`conflicts`
Type	`vpkglist`
Optionality	optional
Default	[]
Description	Intentional representation of packages which conflict with the package being described.

`conflicts` indicate which packages cannot be co-installed, in any given installation, together with a given package. Note that the language to express conflicts is more limited than that used to express dependencies: it consists of plain lists of possibly versioned package names, rather than CNF formulae.

Also note that as far as CUDF is concerned there are no implicit conflicts assumed between different versions of the same package, if they are intended they need to be explicitly specified using the `conflicts` property. According to the CUDF semantics this can be achieved by declaring, for a package p, a conflict with p itself; see Section 2.3 for more information.

Name	`provides`
Type	`veqpkglist`
Optionality	optional
Default	[]
Description	Features provided by the package being described.

A package can declare zero or more *features* that it provides. To abstract over package names, other packages may declare relationships with such features. Packages can provide a specific version of a given feature, or provide a feature without mentioning a version (meaning that *all* possible versions of a given feature are provided by that package).

Name	`installed`
Type	`bool`
Optionality	optional
Default	`false`
Description	Flag stating whether or not the package being described is installed.

Two kinds of packages play a role in the upgrade process: currently installed packages constituting the local package status and (locally or remotely) available packages which are known to

the meta-installer and constitute the package universe. `installed` distinguishes among these two cases, it is `true` for packages which are part of the local status and `false` for those which are part of the package universe. Other kinds of packages that do not play a role in the package upgrade problem are not represented in CUDF.

NAME	`was-installed`
TYPE	`bool`
OPTIONALITY	optional
DEFAULT	`false`
DESCRIPTION	Flag stating whether or not the package being described was installed in a previous package status. Usually this property is only needed when CUDF is used to describe the output of a package solver (see Appendix B); when describing the input for it this property is not needed, but should be considered reserved and not used as an extra package property.

NAME	`keep`
TYPE	`enum`[`version`, `package`, `feature`, `none`]
OPTIONALITY	optional
DEFAULT	`none`
DESCRIPTION	Indication of which aspects of the package being described the user wants to preserve across upgrades. • `version` means preserving the current version, as recorded in the package status. • `package` means preserving at least one version of the package in the resulting package status. • `feature` means preserving all the provided features. • `none` means no particular preservation requirement.

Note that it is not specified *how* the requirements of the `keep` feature have to be fulfilled; in the particular case of `feature` it is possible that the requirement gets fulfilled by replacing a package by some other packages, which, together, provide the same features. See Section 2.3 for the formal specification of the meaning of the `keep` property.

Request description

NAME	`install`
TYPE	`vpkglist`
OPTIONALITY	optional
DEFAULT	[]
DESCRIPTION	List of packages the user wants to be installed.

NAME	`remove`
TYPE	`vpkglist`
OPTIONALITY	optional
DEFAULT	[]
DESCRIPTION	List of packages the user wants to be removed.

NAME	`upgrade`
TYPE	`vpkglist`
OPTIONALITY	optional
DEFAULT	[]
DESCRIPTION	List of packages the user wants to be upgraded to newer versions.

The properties `install`, `remove` and `upgrade` provide the same mechanism for specifying the target packages: lists of package names with optional version specifications. A properly completed `install` action ensures that the requested packages are installed in the resulting package status, on the contrary `remove` ensures that they are not. Since CUDF supports multiple installed versions of the same package – in principle there is no implicit need of removing other packages due to homonym upon `install`. `upgrade` is similar to `install`, but additionally ensures that only one version of each of the target packages is preserved in the resulting packages status; it also ensures that newer versions of them get installed. See Section 2.3 for a formal specification of the semantics of actions.

Extra properties are possible in request description, even though they are not described in this specification. They are allowed as long as their names do not clash with the name of core request properties described above. In particular, we foresee a set of properties that will be used to specify optimization criteria using the forthcoming MOOML language [TZ09a].

2.2.4 Document structure

Putting it all together, the detailed structure of CUDF document is as depicted in Figure 2.3; the figure has to be interpreted as a refined version of Figure 2.1, which we are now able to fill with the properties described in the previous section. Note that all core properties are shown in the figure, in spite of their optionality.

Global constraints

In addition to the per-property constraints reported in the previous section, CUDF documents must respect extra constraints which are not specific to sole information items or properties.

Package/version uniqueness among all package description items forming a given CUDF, there must not exist two package descriptions p_1 and p_2 such that they have the same value of the property `name` and the same value of the property `version`, i.e. the pair of property values $\langle$`name`, `version`$\rangle$ can be used as a "key" to look up package descriptions in a given CUDF.

There is no strict imposition neither in specifying at least one of the `install`/`remove`/`upgrade` properties, nor in specifying non empty-lists as their values. Nevertheless, CUDFs representing problems with empty queries are mostly uninteresting.

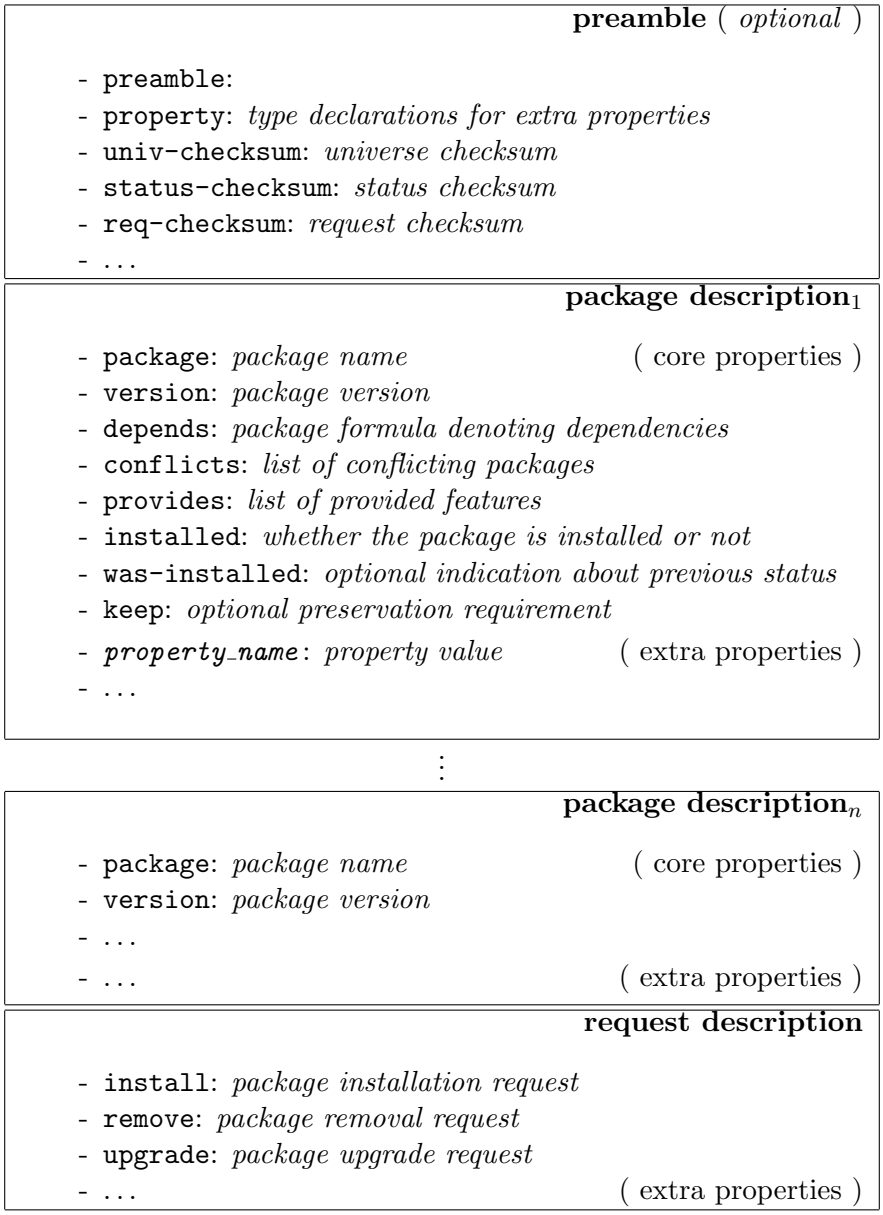

Figure 2.3: Detailed structure of a CUDF document with highlight of core properties.

2.3 Formal semantics

The semantics is defined in a style similar to [MBC+06], however, we now have to deal with an abstract semantics that is closer to "real" problem descriptions, and that contains artifacts like *features*. This induces some complications for the definition of the semantics. In [MBC+06] this and similar problems were avoided by a pre-processing step that expands many of the notions that we wish to keep in the CUDF format.

2.3.1 Abstract syntax and semantic domains

The abstract syntax and the semantics is defined using the value domains defined in Section 2.2.2. In addition, we give the following definitions:

Definition 1
- CONSTRAINTS *is the set of version constraints, consisting of the value* $\top$ *and all pairs* $(relop, v)$ *where* $relop$ *is one of* $=, \neq, <, >, \leq, \geq$ *and* $v \in \mathcal{V}(\texttt{posint})$.
- KEEPVALUES *is the set of the possible values of the* `keep` *property of package information items, that is:* $\{\texttt{version}, \texttt{package}, \texttt{feature}, \texttt{none}\}$

The abstract syntax of a CUDF document is a pair consisting of a package description (as defined in Definition 2) and a request (see Definition 4).

Definition 2 (Package description) *A* package description *is a partial function*

$$\mathcal{V}(\texttt{ident}) \times \mathcal{V}(\texttt{posint}) \rightsquigarrow \mathcal{V}(\texttt{bool}) \times \text{KEEPVALUES} \times \mathcal{V}(\texttt{vpkgformula}) \times \mathcal{V}(\texttt{vpkglist}) \times \mathcal{V}(\texttt{vepkglist})$$

The set of all package descriptions is noted DESCR. *If* ϕ *is a package description then we write* $Dom(\phi)$ *for its domain. If* $\phi(p, n) = (i, k, d, c, p)$ *then we also write*

- $\phi(p, n).\texttt{installed} = i$
- $\phi(p, n).\texttt{keep} = k$
- $\phi(p, n).\texttt{depends} = d$
- $\phi(p, n).\texttt{conflicts} = c$
- $\phi(p, n).\texttt{provides} = p$

It is natural to define a package description as a function since we can have at most one package description for a given pair of package name and version in a CUDF document. The function is generally only partial since we clearly do not require to have a package description for any possible pair of package name and version.

We define the removal operation of a particular versioned package from a package description. This operation will be needed later in Definition 13 to define the semantics of *package conflicts* in case a package conflicts with itself or a feature provided by the same package.

Definition 3 (Package removal) *Let* ϕ *be a package description,* $p \in \mathcal{V}(\texttt{ident})$ *and* $n \in \mathcal{V}(\texttt{posint})$. *The package description* $\phi - (p, n)$ *is defined by*

$$\begin{aligned} Dom(\phi - (p, n)) &= Dom(\phi) - \{(p, n)\} \\ (\phi - (p, n))(q, m) &= \phi(q, m) \qquad \text{for all } (q, m) \in Dom(\phi - (p, n)) \end{aligned}$$

Definition 4 (Request) *A* request *is a triple* (l_i, l_u, l_d) *with* $l_i, l_u, l_d \in \mathcal{V}(\texttt{vpkglist})$.

In a triple (l_i, l_u, l_d), l_i is the list of packages to be installed, l_u the list of packages to be updated, and l_d the list of packages to be deleted.

2.3.2 Installations

Definition 5 (Installation) *An* installation *is a function from* $\mathcal{V}(\texttt{ident})$ *to* $P(\mathcal{V}(\texttt{posint}))$.

The idea behind this definition is that the function describing an installation associates the set of versions that are installed to any possible package name. This set is empty when no version of the package is installed.

We can extract an installation from any package description as follows:

Definition 6 (Current installation) *Let* ϕ *be a package description, the* current package installation of ϕ

$$i_\phi : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$$

is defined by

$$i_\phi(p) := \{n \in \mathcal{V}(\texttt{posint}) \mid (p,n) \in Dom(\phi) \text{ and } \phi(p,n).\texttt{installed} = \texttt{true}\}$$

A package can declare zero or more *features* that it provides. The function f_ϕ defined below associates to any package name (here intended to be a the name of a virtual package) the set of version numbers with which this virtual package is provided by some of the packages installed by ϕ:

Definition 7 (Current features) *Let* ϕ *be a package description, the* current features of ϕ

$$f_\phi : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$$

is defined by

$$\begin{aligned} f_\phi(p) \quad := \{n \in \mathcal{V}(\texttt{posint}) \mid \quad & \text{exists } q \in Dom(i_\phi) \text{ exists } m \in i_\phi(q) \text{ such that} \\ & (((=,n),p) \in \phi(q,m).\texttt{provides} \text{ or } (\top,p) \in \phi(q,m).\texttt{provides})\} \end{aligned}$$

The second case in the definition above expresses the fact that providing a feature without a version number means providing that feature at any possible version.

In order to define the semantics of a CUDF document, we will frequently need to merge two installations. This will mainly be used for merging an installation of packages with an installation of provided features. The merging operation is formalized as follows:

Definition 8 (Merging) *Let* $f, g : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$ *be two installations. Their merge* $f \cup g : \mathcal{V}(\texttt{ident}) \to P(\mathcal{V}(\texttt{posint}))$ *is defined as*

$$(f \cup g)(p) = f(p) \cup g(p) \qquad \text{for any } p \in \mathcal{V}(\texttt{ident})$$

2.3.3 Consistent package descriptions

We define what it means for an installation to satisfy a constraint:

Definition 9 (Constraint satisfaction) *The* satisfaction *relation between a natural number* n *and a constraint* $c \in$ CONSTRAINTS, *noted* $n \models c$, *is defined as follows:*

$$
\begin{array}{lll}
n \models \top & \textit{for any } n & \\
n \models (=, v) & \textit{iff } n = v & n \models (<, v) \quad \textit{iff } n < v \\
n \models (\neq, v) & \textit{iff } n \neq v & n \models (>, v) \quad \textit{iff } n > v \\
 & & n \models (\leq, v) \quad \textit{iff } n \leq v \\
 & & n \models (\geq, v) \quad \textit{iff } n \geq v
\end{array}
$$

Now we can define what it implies for a package installation to satisfy some formula:

Definition 10 (Formula satisfaction) *The* satisfaction *relation between an installation I and a formula p, noted $I \models p$, is defined by induction on the structure of p:*

- $I \models (c, p)$ *where,* $c \in$ CONSTRAINTS *and* $p \in \mathcal{V}(\texttt{ident})$*, iff there exists an* $n \in I(p)$ *such that* $n \models c$.
- $I \models \phi_1 \wedge \ldots \wedge \phi_n$ *iff* $I \models \phi_i$ *for all* $1 \leq i \leq n$.
- $I \models \phi_1 \vee \ldots \vee \phi_n$ *iff there is an* i *with* $1 \leq i \leq n$ *and* $I \models \phi_i$.

We can now lift the satisfaction relation to sets of packages:

Definition 11 *Let I be an installation, and $l \in \mathcal{V}(\texttt{vpkglist})$. Then $I \models l$ if for any $(c, p) \in l$ there exists $n \in I(p)$ with $n \models c$.*

Note that, given that $\mathcal{V}(\texttt{vepkglist}) \subseteq \mathcal{V}(\texttt{vpkglist})$, this also defines the satisfaction relation for elements of $\mathcal{V}(\texttt{vepkglist})$. Also note that one could transform any $l \in \mathcal{V}(\texttt{vpkglist})$ into a formula $l_\wedge \in \mathcal{V}(\texttt{vpkgformula})$, by constructing the conjunction of all the elements of l. The semantics of l is the same as the semantics of the formula $l_\wedge$.

Definition 12 (Disjointness) *The* disjointness *relation between an installation I and a set $l \in \mathcal{V}(\texttt{vpkglist})$ of packages possibly with version constraints, is defined as: $I \parallel l$ if for any $(c, p) \in l$ and all $n \in I(p)$ we have that $n \not\models c$.*

Definition 13 *A package description ϕ is* consistent *if for every package $p \in \mathcal{V}(\texttt{ident})$ and $n \in i_\phi(p)$ we have that*

1. $i_\phi \cup f_\phi \models \phi(p, n).\texttt{depends}$
2. $i_{\phi-(p,n)} \cup f_{\phi-(p,n)} \parallel \phi(p, n).\texttt{conflicts}$

In the above definition, the first clause corresponds to the *Abundance* property of [MBC+06]: all the dependency relations of all installed packages must be satisfied. The second clause corresponds to the *Peace* property of [MBC+06]. In addition, we now have to take special care of packages that conflict with themselves, or that provide a feature and at the same time conflict with that feature: we only require that there be no conflict with any *other* installed package and with any feature provided by some *other* package (see also Section 2.3.5).

2.3.4 Semantics of requests

The semantics of a request is defined as a relation between package descriptions. The idea is that two package descriptions ϕ_1 and ϕ_2 are in the relation defined by the request r if there exists a transformation from ϕ_1 to ϕ_2 that satisfies r. The definition of optimization criteria will is discussed in an independent document [TZ09a].

First we define the notion of a successor of a package description:

Definition 14 (Successor relation) *A package description ϕ_2 is called a* successor *of a package description ϕ_1, noted $\phi_1 \rightarrowtail \phi_2$, if*

1. $Dom(\phi_1) = Dom(\phi_2)$
2. *For all $p \in \mathcal{V}(\texttt{ident})$ and $n \in \mathcal{V}(\texttt{posint})$: if $\phi_1(p,n) = (i_1, k_1, d_1, c_1, p_1)$ and $\phi_2(p,n) = (i_2, k_2, d_2, c_2, p_2)$ then $k_1 = k_2$, $d_1 = d_2$, $c_1 = c_2$, and $p_1 = p_2$.*
3. *For all $p \in \mathcal{V}(\texttt{ident})$*
   - *for all $n \in i_{\phi_1}(p)$: if $\phi_1(p,n).\texttt{keep} = \texttt{version}$ then $n \in i_{\phi_2}(p)$.*
   - *if there is an $n \in i_{\phi_1}(p)$ with $\phi_1(p,n).\texttt{keep} = \texttt{package}$ then $i_{\phi_2}(p) \neq \emptyset$*
   - *for all $n \in i_{\phi_1}(p)$: if $\phi_1(p,n).\texttt{keep} = \texttt{feature}$ then $i_{\phi_2} \cup f_{\phi_2} \models \phi_1(p,n).\texttt{provides}$*

The first and the second item of the above definitions indicate that a successor of a package description ϕ may differ from ϕ only in the status of packages. The third item refines this even further depending on keep values:

- If we have a keep status of `version` for an installed package p and version n then we have to keep that package and version.
- If we have a keep status of `package` for some installed version of a package p then the successor must have at least one version of that package installed.
- If we have a keep status of `feature` for some installed version n of a package p then the successor must provide all the features that where provided by version n of package p.

Definition 15 (Request semantics) *Let $r = (l_i, l_u, l_d)$ be a request. The* semantics *of r is a relation $\overset{r}{\frown} \subseteq$ Descr $\times$ Descr defined by $\phi_1 \overset{r}{\frown} \phi_2$ if*

1. $\phi_1 \rightarrowtail \phi_2$
2. *ϕ_2 is consistent*
3. $i_{\phi_2} \cup f_{\phi_2} \models l_i$
4. $i_{\phi_2} \cup f_{\phi_2} \parallel l_d$
5. $i_{\phi_2} \cup f_{\phi_2} \models l_u$, *and for all p such that $(c,p) \in l_u$ we have that $(i_{\phi_2} \cup f_{\phi_2})(p) = \{n\}$ (i.e., is a singleton set) where $n \geq n'$ for all $n' \in (i_{\phi_1} \cup f_{\phi_1})(p)$.*

2.3.5 Comments on the semantics (non-normative)

Installing multiple versions of the same package The semantics allows a priori to install multiple versions of the same package. This coincides with the semantics found in RPM-like FOSS distributions (which a priori do not forbid to install multiple versions of the same package), but is in opposition to the semantics found in Debian-like FOSS distributions (which allow for one version of any package to be installed at most).

In many practical cases the distinction between a priori allowing or not for multiple versions of a package makes little difference. In the RPM world multiple versions of the same package are very often in a conflict by their features or shipped files. If both versions of the same package provide the same feature and also conflict with that feature then the RPM semantics, as the CUDF semantics, does not allow to install both at the same time. Only packages that have been designed to have distinct versions provide distinct features (in particular, files with distinct paths) can in practice be installed in the RPM world in several different versions at a time. This typically applies to operating system packages. In order to have a meta-installer with Debian semantics work correctly on such a package description, it is sufficient to rename the packages, and to create a new package, say $p - n$, for a package p and version n when p can be installed in several versions.

On the other hand, a meta-installer with RPM semantics will produce solutions on a package description that would not be found by a meta-installer with Debian semantics since it is free to install several version of the same package. The uniqueness restriction of Debian can easily be made explicit in the package description by adding a to each package description stanza, say for package name "p", a serialized property "`conflicts` p".

Upgrading packages Even though the semantics allows for multiple installed versions of the same package, the notion of "upgrade" (at least for what concerns this specification) is intimately tied to a single installed version of a given package.

Hence, for an upgrade request to be fulfilled for a package p, exactly one version of p must be installed in the resulting package status. Additionally, to preserve the "upgrade" intuition, the resulting installed version must be greater or equal than the *greatest* version of p which was previously installed. Both these conditions are expressed by point (5) of Definition 15. Note that a strictly greater version of what was previously installed can be requested by specifying a suitable "`>`" predicate as part of the `upgrade` property.

Upgrading virtual packages Virtual packages, or features, can be with or without version specification. The fact that the lack of version specifications is interpreted as providing all possible versions of a given feature (see Definition 7) interacts with the semantic of upgrades when virtual packages are mentioned within `upgrade`. In particular, upgrades are de facto possible only for versioned virtual packages.[2]

[2]The reason is that upgraded (virtual) packages must correspond to singleton sets in the resulting package status, whereas non-versioned virtual packages will provide infinite sets. Similarly, if in the initial package status a virtual package is non-versioned, it will provide an infinite version sets, whose maximum cannot be matched by any singleton set in the resulting package status.

2.4 Serialization

This section describes how to serialize a CUDF document as a stream of bytes and, symmetrically, how to parse a stream of bytes as a CUDF document. We refer generically to one or the other action as *CUDF serialization*.

Serialization is meant to make the storage of CUDF documents as files possible and to transfer them over the network. A stream of bytes which can be parsed as a CUDF document respecting this specification is called a *CUDF file*.

2.4.1 Overall CUDF file syntax

A CUDF file is a plain-text file containing several *file stanzas*. The bytes composing the file should be interpreted as Unicode characters encoded in UTF-8.

The overall organization of a CUDF file in stanzas reflects the schematic structure of CUDF content (see Section 2.2). Each file stanza is the serialization of a CUDF information item. At least one *blank line* (i.e. empty line, or line composed only by white space characters: `U+0020`, `U+000D`, `U+000A`, `U+0009`) should occur between file stanzas; additional blank lines occurring between file stanzas are ignored. *Comment lines*, i.e. lines starting with the hash character `"#"` in their first column (`U+0023`) are ignored as well.

Serialization must follow the required ordering of information items given in Section 2.2, that is first contain the stanza corresponding to the preamble (if any), then the list of stanzas corresponding to package descriptions (*package description stanzas*, one per package description), and finally the stanza corresponding to request description (*request description stanza*).

To recognize the beginning of file stanzas, each of them starts with a *postmark*, which is specific to information item classes. Postmarks denote the beginning of a new file stanza only *when occurring either at the beginning of the file or just after a newline* (Unicode code point `U+000A`).

- For the preamble item, the postmark is the string `"preamble: "`.
- For package description items, the postmark is the string `"package: "`.
- For the request description item, the postmark is the string `"request: "`.

In both cases, the postmark can be followed by some characters other than a newline, and end with a single newline.

2.4.2 Information item serialization

Each information item, whatever its class, is serialized as a stream of bytes serializing all of its properties in arbitrary order. A single property is serialized as a stream of bytes performing the following steps in order:

1. serialize the property name as the string corresponding to the NAME given in its property schema;
2. output the *name–value separator* string `": "`, i.e. a double colon followed by a space (`U+0020`);

3. serialize the property value;
4. output a single newline.

Let t be the type of a property whose value v has to be serialized as a stream of bytes. The value is serialized by choosing a value from $v' \in \mathcal{L}(t)$ such that $parse_t(v') = v$. That is, all possible values that will be parsed back as the value to be serialized are valid serializations of it.

Since parsing is not one-to-one in general for CUDF types, it is possible that different implementations of this specifications make different choices in terms of how to serialize a given value. Hence it should not be taken for granted that two serializations of CUDF values which are not byte-to-byte identical do not denote the same CUDF value.

An important distinction exists between the serialization of different classes of information items. For package descriptions, the postmark is part of the serialization of properties, i.e. the line starting with `"package: "` is the serialization of the `package` property (i.e. the package name). As a consequence, and in amendment of the general rule above on the property serialization order, ***`package` must be the first property serialized in each file stanza***, because it is used to recognize the beginning of package description file stanzas.

On the contrary, for the preamble and request descriptions the postmark is used to recognize the beginning of the corresponding file stanza, but does not represent the serialization of any particular property. For request descriptions, instead of leaving an empty line after the request description postmark, implementations should output a problem identifier; if the given CUDF serialization has been generated via translation from an available DUDF document [TZ09b], then the problem identifier should be a URI [BLFM98] pointing to the corresponding DUDF document.

A comprehensive example of CUDF file is given in Appendix A.

Line continuations

Sometimes, the serialization of property values can lead to very long strings which can hinder readability of CUDF files. To counter that problem, CUDF serialization allow to split the serialization of long values across multiple lines.

In fact, each *logical property line* in a file stanza can be serialized over multiple *physical lines* composed by a starting property line and followed by 0 or more *continuation lines*. Each continuation line starts with a space character (`U+0020`) in its first column. During serialization (or, symmetrically, during parsing) such a list of lines is considered equivalent to a single physical line obtained as follows: start with the first property line and remove its trailing newline, concatenate the next continuation line by first removing its heading space character, remove its trailing newline, and continue until continuation lines are finished.

For instance, the long value "`pin-priority: int , suite: enum[stable,testing,unstable] = [stable]`" of `property` seen in Section 2.2.2 can be output on multiple physical lines (a property line and a continuation) obtaining the following property serialization snippet:

```
property: pin-priority: int ,
 suite: enum[stable,testing,unstable] = [stable]
```

After parsing, a single `property` will be returned; its value will be obtained by joining the above two physical lines as discussed above.

2.4.3 CUDF file parsing

How to parse CUDF files to obtain CUDF documents is almost straightforward and follows from an analysis of the serialization rules given above.

Parsing errors can be encountered while parsing CUDF serializations which do not match the rules provided by this specification. Parsing errors can be localized at specific positions of the CUDF serialization. When the position of a parsing error belongs to a specific file stanza (i.e. it is in between two postmarks, or between a postmark and the end of file), that error is said to be recoverable. The recovery strategy is to ignore the file stanza the error belongs to and act as if that stanza was not there.

The actual parsing procedure is as follows:

1. Given a CUDF file, split it at occurrences of postmarks. The result of this operation is a list of file stanzas. Each of them can be recognized as the serialization of either the preamble (if the postmark is `"preamble: "`, a package description (postmark `"package: "`), or a request description (postmark `"request: "`).

   Afterwards, preamble and request description postmarks are useless and can be ignored for further processing. On the contrary, package description postmarks should be integrated again as part of the following package description file stanza.

2. Parse each file stanza as a list of property serializations by splitting at occurrences of newlines.

3. Parse each property serialization as a pair of property name and value serializations by splitting when the string `": "` occurs. If needed, handle line continuations (see Section 2.4.2) in this phase.

4. Turn each property name serialization into a property name in a straightforward way, as long as it matches the constraints on property names. Otherwise raise a parsing error; the error must be located in the file stanza owning the affected property.

   For each property name check whether that property is supported by the information item serialization in which it appears. If this is the case then this specification permits to assign all the information coming from its schema to that property, in particular a type and possibly a default value. If the property is not supported by this specification for a given information item, it is either known, via some unspecified external mechanism, how to associate a schema to that property or that property cannot be processed any further and will be disregarded.

   After this step all properties have an associated schema and a (yet to be parsed) value serialization.

5. For each value serialization parse it using the parsing function of the associated property type. If the function is not defined for the given serialization then a parsing error is raised; the error is located in the file stanza owning the affected value.

   After this step, each file stanza has been parsed into a list of properties as supported by CUDF. That list can be turned into a set. If the same property name appears twice or more in the set, a parsing error is raised; the error is located in the file stanza containing the properties.

   Once sets are formed, the CUDF file has been fully parsed into a list of information items; together they already form a CUDF document.

6. The only missing step is handling of default values. For each information item check whether some of the optional properties are missing according to the information item kind. For each such property, add a property of that name to the information item where it was missing. The corresponding value is the default value defined in the property schema.

Compatibility with RFC822 (non-normative)

Conforming implementation of CUDF serialization produces CUDF files which are blank-separated sequences of messages conforming to RFC822 [Cro82]. The only exception to this rule is that comment lines should be removed from CUDF serializations before obtaining RFC822-compatible documents.

This aspect hints an alternative—yet correct—way of parsing CUDF files via exploitation of existing RFC822 implementations. On top of them it is enough to perform the parsing steps given above from 4 to 6, skipping steps 1–3 which are subsumed by RFC822 parsing.

2.5 Generating CUDF (non-normative)

While it is possible to generate CUDF documents directly, it is expected that the largest fraction of the CUDF corpus to be used for the competition will be generated via conversion from DUDF documents provided by users of FOSS distributions. The format of DUDF documents is described in [TZ09b].

Each distribution interested in providing upgrade problems for the UPDB (see Chapter 1) is then required to provide specification and tools that implement the conversion. Ideally, the description of how to convert from a specific DUDF instance and CUDF should be described together with the specification of the specific DUDF instance. It is expected that each partner interested in contributing problems to the UPDB publishes a document describing both aspects.

During the conversion, we expect three main tasks to be implemented.

Translation: package information $\longrightarrow$ package information items Each DUDF instance is expected to encode the information about all packages known to the meta-installer in some way. The first required task to create the resulting CUDF is to convert such (meta-installer-/distribution-specific) information to package information items as described in this specification.

The implementation of this task should account not only for data conversions imposed by the CUDF type system (e.g. translating from legacy versioning schemata—$x.y.z$—to positive integers), but also for semantic differences between the origin distribution and CUDF. Likely, the most common cause of semantic incompatibilities will be the translation from Debian-like packaging systems to CUDF (see Section 2.3.5 for advice on how to address this problem).

Translation: user request $\longrightarrow$ request description item Similarly, the request that the user posed to its meta-installer needs to be translated to a request description item.

The request language supported by CUDF is expected to be expressive enough to encode the vast majority of user requests nowadays expressible in state of the art meta-installers.

Exceptions are of course possible, in which case no translation from DUDF to CUDF is possible. Specifications of DUDF instances must clearly state such limitations.

Expansion of intentional sections DUDF encodings are expected to be more compact than the corresponding CUDF encoding. To that end DUDF documents can refer to external resources whereas CUDF documents are expected to be entirely self-contained. Therefore, all references to external entities occurring in DUDF documents must be expanded before being able to create the corresponding CUDF encoding.

Since in general only distributions are expected to be able to perform the expansions (e.g. because the referred repositories are mirrors or databases hosted by them), the actual translation from DUDF instances to CUDF should be performed by distributions *before* injecting problems into the central UPDB.

Serialization Once all information items translated from DUDF to the CUDF model, they need to be serialized to files (see Section 2.4).

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 defines the CUDF (Common Upgrade Description Format) which plays a central role in the Mancoosi infrastructure. The purpose of CUDF 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 upgrade scenarios 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-contained problem.

As far as CUDF is concerned, this specification has provided a formal model in which constraints coming from popular packaging "worlds" (e.g. Debian and RPM) can be expressed. On top of that model the semantics of typical upgrade action requests (e.g. install, remove, upgrade) has been described; using that semantics it is possible to check whether a solution provided by a solver properly implements a given user request.

In addition to the formal model, this specification has also provided a document structure in which both the user request and the universe of all packages known to a package manager can be encoded. Parsing and serialization rules for the document structure have been given as well, so that CUDF documents can be dealt with in file form. Solvers taking part in the competition are meant to parse CUDF files in order to obtain the upgrade problem they are asked to solve.

To complete the competition scenario two important aspects are missing, but have been left beyond the scope of this document on purpose:

Optimization criteria It is expected that solvers taking part into the competition will not simply be asked to solve a given upgrade problem. At least for some competition "tracks", there will be extra requirements to find the best possible solution among several alternative solutions which are correct according to the CUDF model.

How to specify optimization criteria is beyond the scope of this document. Each competition track will advertise the optimization criteria to be implemented by participating solvers. Optimization criteria can be defined on top of package properties which are already expressible in the present version of CUDF. To this end, CUDF is extensible: additional properties not prescribed by this specification can be added to package descriptions, by exploiting existing CUDF types. Similarly, when there is a need to encode *within* CUDF optimization criteria, that will be possible (at the document structure level and without violating any CUDF assumption) by using extra properties in the `"request"`

stanza.

A language to specify optimization criteria will be published in a separate document.

Solver output format The output format of solvers taking part in the competition is beyond the scope of this specification. Nevertheless it will be needed in order to have a common way to understand the solutions found by solvers and to determine their quality according to the optimization criteria.

Straightforwardly, the solver output can be encoded by serializing the new local package status as if it were a CUDF document missing the problem description item. An output format based on this approach is further discussed in Appendix B.

Appendix A

CUDF serialization example (non-normative)

This non-normative section contains an example of CUDF serialization to file. The example below has been inspired by the EDOS car/glass example [EDO06].

Some remarks about the example follow.

- The example uses extended properties.
- Intuitively, the example comes from a packaging world where different versions of the same package are implicitly conflicting with each other. To grasp this, all packages for which multiple versions are available declare a non-versioned conflicts with themselves.
- The `engine` feature is mutually exclusive, only one (installed) package can provide it. This is encoded using conflicts with the feature from each package providing it.

```
preamble:
property: suite: enum[stable,testing,unstable] = [stable],
 bugs: int = [0],
 description: string = ["no description"]
univ-checksum: 8c6d8b4d0cf7027cd523ad095d6408b4901ac31c
status-checksum: 6936ce910eb716ad97190393f80c14ab04d95b3d
req-checksum: 17259225eaf63642f9ab99a627b9857a5b27c5f7

package: car
version: 1
depends: engine, wheel, door, battery
installed: true
description: 4-wheeled, motor-powered vehicle

package: bicycle
version: 7
description: 2-wheeled, "pedal-powered" vehicle

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
suite: testing

package: electric-engine
version: 1
depends: solar-collector | huge-battery
provides: engine
conflicts: engine, electric-engine
bugs: 12

package: electric-engine
version: 2
depends: solar-collector | huge-battery
provides: engine
conflicts: engine, electric-engine
suite: unstable
bugs: 180

package: solar-collector
version: 1

package: battery
version: 3
provides: huge-battery
installed: true

package: wheel
version: 2
conflicts: wheel
installed: true
suite: stable

package: wheel
version: 3
conflicts: wheel
suite: testing

package: door
version: 1
conflicts: door
installed: true

package: door
version: 2
depends: window
conflicts: door
suite: unstable

package: turbo
version: 1
installed: true
```

```
package: tire
version: 1
conflicts: tire

package: tire
version: 2
conflicts: tire
suite: testing

package: window
version: 1
conflicts: window

package: window
version: 2
depends: glass = 1
conflicts: window
suite: testing

package: window
version: 3
depends: glass = 2
conflicts: window
suite: unstable

package: glass
version: 1
conflicts: glass

package: glass
version: 2
conflicts: glass, tire = 2
suite: testing

request: http://www.example.org/8f46e388-042f-415e-8aab-df4eeb974444.dudf
install: bicycle, electric-engine = 1
upgrade: door, wheel > 2
```

Appendix B

Solver output format (non-normative)

This non-normative section shows how to use the CUDF document format to encode the output of solvers; assuming their input is encoded in CUDF, that offers a complete, pure-CUDF API for package solvers. The format described here is called *CUDF output format*.

The output of a solver must contain all information needed to change the package status which was in effect before the solver was invoked, bringing it to a new package status found by the solver. Since the package universe is not allowed to change during solving (see Definition 14), it is enough to encode in the solver output the new package status, i.e. all and only those packages that according to the solver must be installed to satisfy user request.

Such result is achieved in the CUDF output format as a CUDF-like document that contains *only package description stanzas* and where all packages that the solver wants to be installed have corresponding stanzas with the `installed` property set to `true`.

Extra package description stanzas for non-installed packages can be output as well, but are not needed to determine the obtained package status. If there is a need to explicitly show what changed between solver input and output, the property `was-installed` can be used to denote whether a given package was installed before solver action.

With the exception of `installed`, all non-required package properties are not needed to determine the new package status (since they are not allowed to change during solving as well) and can hence be dropped to have a terse output. Required properties, in particular package name and version, must be present in each package description stanza.

In CUDF output format the optional preamble stanza is allowed—as that would permit type-checking on the solver output—whereas the request description stanza is forbidden.

All rules for CUDF documents that are not related to the request description stanzas apply to CUDF output format documents, in particular serialization rules are the same.

A sample serialization of a CUDF output format is given below:

```
preamble:
property: suite: enum[stable,testing,unstable] = [stable]
property: bugs: int = 0

package: bicycle
version: 7
installed: true
was-installed: false
```

```
package: electric-engine
version: 2
installed: true
was-installed: false

package: solar-collector
version: 1
installed: true
was-installed: false
```

All universe packages not listed in the output are considered not to be installed in the solution proposed by the solver; if they were previously installed they must be removed to implement the proposed solution.

Appendix C

Changes from previous versions

C.1 From deliverable D5.1 (i.e. "CUDF 1.0") to CUDF 2.0

Major changes

- Chapter 2 (several changes, throughout the chapter): add a new preamble information item, its main purpose being declaring in advance extra package properties together with their types and (optional) default value

Noteworthy changes

- Section 2.2
  - make mandatory the specification of a default value for optional properties. As a consequence: change the type of the `keep` property to allow an explicit `'none` value, and use it in the formal semantics.
  - require that enumeration types must contain at least one value.
  - better engineering of identifier-like lexical conventions: distinguish between *identifiers* (for property names and enumerations) and *package names*. As a consequence, property names are now all lowercase
  - fix the ordering of stanzas in CUDF, drop the notion of (only recommended) "canonical ordering"; change propagated to Section 2.4.
- Chapter 2 (various changes, throughout the chapter): rename the "*problem* description item" to "*request* description item". Propagate the change also to postmark names: from `problem:` to `request:` in CUDF serialization
- Definition 7: this function associates now to any name of a feature the set of version numbers with which this feature is realized by the current installation. Removed the definition of expand-features which was obsolete.
- Definition 15: account for upgrades of virtual packages. To better explain that notion, also added 2 related new paragraphs in non-normative Section 2.3.5 "Comment on the semantics".
- Type library, Section 2.2.2:
  - be more liberal in package names:
    - allow the following extra characters in package names (type `pkgname`): `"+"`, `"/"`, `"%"`, `"@"`, `"("`, `")"`, uppercase Latin letters

    - allow package names to start with any of the allowed characters
    - allow single-character package names, e.g. package "`p`"
  - extend value and lexical spaces of `vpkgformula` to allow writing `true!` and `false!` values explicitly.
  - change the syntactic representation of `enum` types: use square brackets instead of round parentheses
- CUDF serialization (Section 2.4):
  - allow shell-like comments: all lines starting with `"#"` in their first column are ignored
  - allow RFC2882-like line-continuations to split long (logical) lines across several (physical) lines (Section 2.4.2)
  - require at least one blank line between file stanzas
- new Appendix B: specification of the *CUDF output format*
- Core properties (Section 2.2.3):
  - add new core preamble properties `univ-checksum`, `status-checksum`, `req-checksum`
  - add new core package property `was-installed`

Minor fixes

- Appendix A: add missing "`Provides: engine`" on package "`gasoline-engine`"
- Section 2.2.2:
  - lexical space of `vpkg`:
    - the optionality of spaces separating PkgName from its VConstr must be synchronized with VConstr (i.e. spaces are not allowed if the version constraint is missing)
    - remove extra (and useless) space non-terminals
    - allow tabs as "blanks" as well as spaces in structured types (e.g. `vpkg`)
  - lexical space of `vpkgformula`: add a missing trailing `*` in the OrFla non-terminal
  - convention for typesetting enumeration values in the spec: drop the leading `'` as it was useless
- Definition 8: fixed trivial typo (Merging of functions).
- Section 2.4.3: fix typo in the description of what to do with package postmark after recognizing the begin of a package stanza (it should be kept)
- Section 2.2: make explicit that property names are unique in any given information item
- The section on mathematical optimization criteria has been removed from this document since optimization criteria are the subject of the forthcoming report on the MooML language.
- Suggest to use URIs pointing to corresponding DUDF documents after the `request:` postmark.

Bibliography

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. RFC 2396 (Standards Track), August 1998.

[BM04] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes Second Edition*. http://www.w3.org/TR/xmlschema-2, October 2004. W3C Recommendation.

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

[Con06] The Unicode Consortium. *Unicode Standard, Version 5.0, The*. Addison-Wesley Professional, 5 edition, November 2006.

[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.

[EBN96] Extended BNF. ISO/IEC 14977 (International Standard), 1996.

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

[MBC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006*, pages 199–208, Tokyo, Japan, September 2006. IEEE CS Press.

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, 1st edition, October 2003.

[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.

[TZ09a] Ralf Treinen and Stefano Zacchiroli. Expressing advanced user preferences in component installation. In Roberto Di Cosmo and Paola Inverardi, editors, *IWOCE '09: Proceedings of the 1st international workshop on Open component ecosystems*, pages 31–40. ACM, August 2009.

[TZ09b] Ralf Treinen and Stefano Zacchiroli. Upgrade description formats: generalities and DUDF submission format. Technical Report TR001, The Mancoosi project, September 2009. http://www.mancoosi.org/reports/tr1.pdf.