Dependency Solving: a Separate Concern in Component Evolution Management[☆]

Pietro Abate[a], Roberto Di Cosmo[a,b], Ralf Treinen[a], Stefano Zacchiroli[a]

[a] *Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France*
[b] *INRIA Paris-Rocquencourt, F-75205 Paris, France*

Abstract

Maintenance of component-based software platforms often has to face rapid evolution of software components. Component *dependencies*, *conflicts*, and *package managers* with *dependency solving* capabilities are the key ingredients of prevalent software maintenance technologies that have been proposed to keep software installations synchronized with evolving component repositories.

We review state-of-the-art package managers and their ability to keep up with evolution at the current growth rate of popular component-based platforms, and conclude that their dependency solving abilities are not up to the task.

We show that the complexity of the underlying upgrade planning problem is NP-complete even for seemingly simple component models, and argue that the principal source of complexity lies in multiple available versions of components. We then discuss the need of expressive languages for user preferences, which makes the problem even more challenging.

We propose to establish dependency solving as a *separate concern* from other upgrade aspects, and present *CUDF* as a formalism to describe upgrade scenarios. By analyzing the result of an international dependency solving competition, we provide evidence that the proposed approach is viable.

Keywords: component, dependency solving, software evolution, package management, open source, competition

1. Introduction

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful.

[☆] This work has been partially performed at IRILL http://www.irill.org

Email addresses: `abate@pps.jussieu.fr` (Pietro Abate), `roberto@dicosmo.org` (Roberto Di Cosmo), `treinen@pps.jussieu.fr` (Ralf Treinen), `zack@pps.univ-paris-diderot.fr` (Stefano Zacchiroli)

URL: `http://www.pps.jussieu.fr/~abate` (Pietro Abate), `http://www.dicosmo.org` (Roberto Di Cosmo), `http://www.pps.jussieu.fr/~treinen/` (Ralf Treinen), `http://upsilon.cc/~zack` (Stefano Zacchiroli)

The above law of *Continuing Change* [20] applies to all evolving software systems, which are deemed to be the vast majority of existing systems [7]. The advent of Component-Based Software Engineering [6, 36] did not affect this fundamental truth: *mutatis mutandis* continuing change also holds for component-based systems [21]. The diffusion of rapidly evolving *component-intensive software platforms*—i.e. platforms where the number of components is in the tens or even hundreds of thousands—has raised the quality requirements for automatic tools that maintain component installations on behalf of users, be them developers, architects, administrators, or final users empowered to assemble components.

Component-intensive platforms are commonplace: FOSS (Free and Open Source Software) distributions (where components are called "packages"), development platforms like Eclipse and Apache Maven [9, 25] (which call components "plugins"), OSGi [29] ("bundles"), CMS communities ("add-ons"), Web browsers ("extensions"), and countless others. Despite apparent differences in terminology, all these platforms share concepts, properties, and problems. For instance, components have expectations on the deployment context: they may need other components to function properly—declaring this fact by means of *dependencies*—and may be incompatible with some other components—declaring this fact by means of *conflicts*. Those expectations must be respected not only at initial deployment-time, but also at each component release and for each individual component: a new version of a component cannot be deployed if its expectations are not met on the target system.

To maintain component assemblies, (semi-)automatic component manager applications are used to perform component installation, removal, and upgrades on target machines—we use the term *upgrade* to refer to any combination of those actions. Examples of component managers are as commonplace as component-intensive platforms: package managers, such as APT or Aptitude used in FOSS distributions to manage packages; P2 [19], used in Eclipse to deal with plugins; OSGi resolvers, which perform component deployment and configuration. These tools—called generically *package managers* in the following—incorporate numerous functionalities: trusted retrieval of components from remote repositories; planning of upgrade paths in fulfillment of deployment expectations (also known as *dependency solving*); user interaction to allow for interactive tuning of upgrade plans; and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered at deploy-time [10].

In contexts where the pace of component releases is rapid (e.g. FOSS [31, 14, 1]) the quality demand on package managers, and in particular on dependency solving, is very high. Package managers should: (1) devise upgrade plans that are correct (i.e. no plan that violates component expectations is proposed) and complete (i.e. every time a suitable plan exists, it can be found); (2) have performances that scale up gracefully at component repositories growth; (3) empower users to express preferences on the desired component configuration when several options exist, which is often the case. Surprisingly, all mainstream component manager applications the authors are aware of fail to address one or several of those concerns. Not addressing them is far from being a purely academic exercise, as Figures 1 and 2 show. Although anecdotal those and similar examples, which populate the experience of everyday package manager users, show that state-of-the-art component managers are short of fulfilling the aforementioned requirements.

```
# aptitude upgrade
1163 packages upgraded, 633 newly installed,
195 to remove and 0 not upgraded.
The following packages have unmet dependencies:
[...]
open: 4892; closed: 4995; defer: 170; conflict: 86
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 7592; closed: 7654; defer: 193; conflict: 89
open: 7798; closed: 7879; defer: 233; conflict: 89
open: 9938; closed: 9977; defer: 315; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 14915; closed: 14952; defer: 372; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 19880; closed: 19981; defer: 445; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 25017; closed: 24998; defer: 467; conflict: 90
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 30110; closed: 29978; defer: 498; conflict: 91
No solution found within the allotted time. Try harder? [Y/n]n
```

Figure 1: Unexpected behaviour while using the legacy Aptitude package manager, on a FOSS system on the Debian GNU/Linux distribution. The user attempts to upgrade all components in need of upgrade on a machine equipped with the GNOME desktop environment and several LaTeX packages. The dependency solver loops and is unable to find a solution; after several attempts, the user gives up. (See http://bugs.debian.org/590470; retrieved November 29th, 2010.)

Considering the recent popularity of dependency-based abstractions in Component Based Software Engineering (CBSE, e.g. [17, 33, 11]), overlooking important dependency solving requirements appears to be dangerous.

This work provides substantial coverage of concepts and problems that are common in component managers equipped with automatic dependency solving abilities, for any non-trivial component model. Understanding such problems is of paramount importance because, in the context of component-intensive software platforms, software evolution is observed by users through the lens of component releases and often judged by the package manager abilities to successfully deploy new releases. Therefore, to avoid software evolution bottlenecks at the component deployment stage, we need to improve the ability of our tools to plan component upgrades. Unfortunately, as we will show, the problem is a hard one to tackle. In order to attack such a non-trivial and fairly overlooked problem, this paper proposes to treat dependency solving as a separate concern of component evolution and details the formalisms and technologies that are needed to enable such separation.

Paper contributions and structure. In Section 2 we present the upgrade planning problem, or simply *upgrade problem*, in a general setting, showing that in any non-trivial component model dependency solving is NP-complete. To tackle the problem, in Section 3 we propose to treat dependency solving as a separate concern, in order to share research and development efforts on upgrade planning. To that end, we need formalisms to: (1) capture upgrade scenarios coming from different component models in a unifying, well-defined semantics and (2) describe user preferences which are advanced enough to

```
# aptitude install baobab
[...]
The following packages are BROKEN: gnome-utils
The following NEW packages will be installed: baobab
[...]
The following actions will resolve these dependencies:
Remove the following packages:
  gnome gnome-desktop-environment libgdict-1.0-6
Install the following packages:
  libgnome-desktop-2 [2.22.3-2 (stable)]
Downgrade the following packages:
  gnome-utils [2.26.0-1 (now) -> 2.14.0-5 (oldstable)]
[...]
0 packages upgraded, 2 newly installed, 1 downgraded,
180 to remove and 2125 not upgraded. Need to get 2442kB
of archives. After unpacking 536MB will be freed.
Do you want to continue? [Y/n/?]
```

Figure 2: Attempt to install a disk space monitoring utility (called *baobab*) using Aptitude. In response to the request, the package manager proposes to downgrade the GNOME desktop environment all together to a very old version compared to what is currently installed. As shown in Section 6 a trivial alternative solution exists that minimizes system changes: remove a couple of dummy (or "meta") packages.

cover realistic use cases, but yet simple enough to be efficiently dealt with by state-of-the-art constraint solvers. Our proposals for those two formalisms are detailed in Sections 4 and 5. Section 6 validates the proposed approach by discussing an international dependency solving competition—called MISC—which has been run exploiting the proposed formalisms. Competition results show that state-of-the-art constraint solvers can easily outperform ad-hoc solvers embedded in mainstream package managers, confirming the thesis that separation of concerns and reuse are not only feasible, but also a viable strategy to improve upgrade planning and support component evolution.

2. Component Evolution and the Complexity of the Upgrade Problem

In this section we start by studying the complexity of the *upgrade problem* that package managers for component-intensive software platforms have to face. An important feature of the problem is that there is usually a multitude of possible choices. This has two consequences:

- For any given user request, there potentially exists an exponential number of solution candidates, which makes the problem NP-complete in all relevant cases (see Sections 2.1 and 2.2).
- There might be an exponential number of actual solutions to a problem instance, and we need a good way to pick the best among these solutions (see Section 2.3).

2.1. Complexity of the Upgrade Problem

A *software component* is a bundle of: (1) files that are to be installed on a target machine, (2) configuration logic to be executed at various stages of deployment, and (3) metadata which, among others, describe component expectations [10]. For the purpose of this paper we focus only on metadata since this is the information used by package managers to plan upgrades. There are different component models, but metadata contains at least the following features:

name: a component identifier that has a meaning over a time-line of releases;

version: an identifier of a specific release of a component that is meaningful relative to a given name;

dependencies: components that must be installed to make a component usable.

The expressiveness of the dependency language varies, but at the very minimum allows for a list of components that are required to be installed. More evolved models also allow for disjunctions (alternatives) and version constraints (like "component c in any version greater than 42"). Most component models also allow for:

conflicts: components that are not to be installed at the same time as the given component. Conflicts may come with version constraints, similar to dependencies.

features: names of virtual components *provided by* a component. They may be used to satisfy dependencies of other components and must not conflict with other installed components.

We assume that each package is uniquely identified by its name n and version v, and denote it as (n, v). A *repository* R is a set of components. An *R-installation* I is a set of components $I \subseteq R$ that has the properties of:

abundance: each package in I has its dependencies satisfied by packages in I;

peace: no package in I conflicts with another package in I.

The following theorem was proven, in a more specific context, in [12]:

Theorem 1. *Satisfiability of package upgrade requests is NP-complete, provided the component model features conflicts and disjunctive dependencies.*

Proof. First, we remark that the problem is clearly in NP since, given a subset of the repository, one can check in polynomial time that it satisfies abundance, peace, and the specific user request.

To prove NP-completeness, we show how to reduce the well known NP-complete problem 3-SAT to the upgrade problem. For this, we show that any instance of 3-SAT can be encoded into a simple instance of the upgrade problem, consisting of a single component installation request in an empty initial installation.

Let $F = C_1 \wedge \ldots \wedge C_n$ be an instance of the 3-SAT problem, where each C_i is a disjunction of three literals. We define a repository R_F that contains:

- for every literal L occurring in F a package $(L, 0)$ which conflicts with the package whose name is the complement of L,
- for every clause $C_i = L_i^1 \vee L_i^2 \vee L_i^3$ occurring in F a package $(C_i, 0)$ which depends on the disjunction of the packages $(L_i^1, 0)$, $(L_i^2, 0)$, and $(L_i^3, 0)$,
- a package $(F, 0)$ which depends on the conjunction of the packages $(C_1, 0), \ldots, (C_n, 0)$.

It is easy to see that F has a solution iff there is an R_F installation containing package $(F, 0)$. Note that no sophisticated usage of versions is needed for this encoding: we have used version 0 everywhere. □

The above proof makes essential use both of disjunctions in dependencies, and conflicts. In fact there are different ways how disjunctions in dependencies may appear: through explicit alternatives (as used in the proof), features, or multiple versions of a package. In fact, having both conflicts and disjunctions (in any form) are crucial for NP-completeness, as the following theorem shows:

Theorem 2. *Installability of a package in an empty environment is in PTIME in any of the two following cases:*

1. *The component model does not allow for conflicts.*
2. *The component model does not allow for disjunctive dependencies or features, and the repository does not contain multiple versions of packages.*

Proof. We first recall that component installability can be encoded into Boolean satisfiability [23]. Given a repository R, we construct a logical theory T_R as follows: we introduce, for every component or feature in R of name n and version v, a propositional variable X_n^v. A dependency d of package (n, v) is translated into an implication $X_n^v \to \bar{d}$, where $\bar{d}$ is the logical formula representing the dependency d, obtained by replacing an atomic dependency by the disjunction of all variables corresponding to components satisfying that atomic dependency[1]. For every conflict (n', v') of a package (n, v) we add a formula $\neg X_n^v \vee \neg X_{n'}^{v'}$. If packages $(n_1, v_1), \ldots, (n_k, v_k)$ provide feature f of version v then we add $X_f^v \to (X_{n_1}^{v_1} \vee \ldots \vee X_{n_k}^{v_k})$. It is easy to see that $T_R \wedge X_n^v$ has a propositional model iff there exists an R-installation that contains component (n, v). The formula $T_R \wedge X_n^v$ falls into particular classes in the two cases of the theorem:

1. If there are no alternatives in dependencies, no features, and no multiple versions of packages then all implications obtained from dependencies are of the form $X \to (X_1 \wedge \ldots \wedge X_n)$, which is equivalent to $(X \to X_1), \ldots, (X \to X_n)$. Since clauses obtained from conflicts are always binary, and since the formula X_n^v is unary, one obtains a theory which is a set of unary and binary clauses. The PTIME results follows since satisfiability of sets of unary and binary clauses is decidable in polynomial time [32].
2. If there are no conflicts then one just has formulas $(X_n^v \to \bar{d})$. Since all occurrences of literals in $\bar{d}$ are positive, we can rewrite each of these formulas by transforming $\bar{d}$ into disjunctive normal form as a set of clauses of the form $X_n^v \to (L_1 \vee \ldots \vee L_n)$. These are *dual Horn clauses*, that is clauses that contain at most one negative literal. Satisfiability of sets of dual Horn clauses is again decidable in PTIME ([32], who calls them *weakly positive clauses*).

□

2.2. Complexity in the Case of Component Evolution

The problem of package installation becomes significantly harder when one imposes that old versions of packages have to be *replaced* by new versions of packages, instead of just installing old and new version at the same time. This requirement appears in different form in different component models:

[1] This disjunction is empty, yielding the formula $\bot$, in case the package mentioned in the dependency is absent from the repository.

- The Debian package model allows to install only one version of a package at a time.
- In the RPM package model, it is a priori possible to install multiple versions of a package at a time; however it is in practice almost always excluded by the fact that different versions of a package install files with the same path on the file system, and hence are in conflict with each other.
- The Eclipse model allows for an explicit *singleton* property in component meta-data, with the semantics that only one version of that component must be installed.

In order to state a complexity result, we will consider in this subsection that component installations must be **flat**, that is must not contain two packages with the same name (which then would have different version). The complexity result stated in the following theorem, however, applies equally to the other models mentioned above since we may always require uniqueness of version for the packages used in the proof.

Theorem 3. *Existence of a* flat *installation containing a component is NP-complete, even when the component model does not allow for explicit conflicts, alternatives, and features.*

Proof. The problem is in NP for the same reason as in Theorem 1: one can check in polynomial time for every subset of the repository whether it satisfies abundance, peace, flatness, and the specific user request.

We show NP-completeness by giving a polynomial reduction of the 3-SAT problem. Let $F = C_1 \wedge \ldots \wedge C_n$ be a problem instance, where each C_i is of the form $C_i = L_i^1 \vee L_i^2 \vee L_i^3$. We define a repository R_F consisting of the following components:

- for each propositional variable X a package with name X, existing in versions 0 and 1. Each of these versions has no explicit conflicts or dependencies.
- for each clause C_i a package C_i in three versions 1, 2, and 3. None of them has conflicts. If the literal L_i^j $(j = 1, 2, 3)$ is a positive literal X then component (C_i, j) depends on $X(= 1)$. If the literal L_i^j is a negative literal $\neg X$ then component (C_i, j) depends on $X(= 0)$.
- a package of name F and version 1 that depends on $C_1, \ldots, C_n$.

If there is a flat R_F-installation containing $(F, 1)$ then F is satisfiable : Any flat installation may in particular contain at most one version of any package associated to a propositional variable. Hence, a flat R_F-installation I defines a propositional valuation α_I (if I does not contain any version of a package X then we may choose $\alpha_I(X)$ arbitrarily), and when I contains $(F, 1)$ then α_I obviously satisfied the 3-SAT instance F.

If F is satisfiable then there exists a flat R_F-installation containing $(F, 1)$: Let α be a solution of F. This means that one may choose, for any clause C_i, one index $s(i) \in \{1, 2, 3\}$ such that α satisfies the literal $L_i^{s(i)}$. We construct an R_F-installation from all the packages corresponding to propositional variables in the version according to their respective truth value in α, the packages $(L_1, s(1)), \ldots, (L_n, s(n))$, and finally the package $(F, 1)$. □

In some sense, a dependency on a package with name n acts like an exclusive choice in case of the flatness requirement on installations. If we have versions 1, 2 and 3 of packages with name n, then an unqualified dependency on name n can be read as the requirement on exactly one of $(n, 1)$, $(n, 2)$, $(n, 3)$.

For the problem to be NP-complete, it is enough to have just two versions of each component:

Corollary 1. *Theorem 3 holds for repositories containing at most two versions per package.*

Proof. It is sufficient to replace in the above proof each of the components C_i by two components, C_i^1 and C_i^2, each of them coming in version 1 and 2:

- $(C_i^1, 1)$ depends on (X, v) corresponding to the first literal of C_i,
- $(C_i^1, 2)$ depends on C_i^2,
- $(C_i^2, 1)$ depends on (X, v) corresponding to the second literal of C_i,
- $(C_i^2, 2)$ depends on (X, v) corresponding to the third literal of C_i,

The component $(F, 1)$ depends on $C_1^1, \ldots, C_n^1$. □

2.3. Dealing with exponentially many solutions

Having established the complexity of finding *a* solution to an upgrade problem, we now turn our attention to *the amount* of existing solutions for any given user request. The interest in analyzing that aspect stems from the observations that, among all possible solutions, package managers generally try to offer to the user the "best" solution, at least according to some predefined strategy. Indeed an often overlooked fact is that a user request that consists of just a list of components to install, remove or upgrade may have exponentially many solutions. This is closely related to the complexity results of the previous section which rely on the fact that there is an exponential number of solution *candidates*.

Example 1. *Consider a repository R consisting of components q_i, for $1 \leq i \leq n$, in versions 1 and 2, and a component p in version 1 depending on all of $q_1, \ldots, q_n$ in any version. The initial installation contains each of the package q_i in version 1, and we ask to install package p, where installations have to be flat.*

Any of the 2^n configurations $\{(p, 1)\} \cup \{(q_i, i) | i \in 1 \ldots n, 1 \leq i \leq 2\}$ is a solution.

These 2^n solutions are all pretty different from a user point of view. The solution that keeps the originally initially version of all the q_i may be preferred by "*paranoid*" users who want to avoid unnecessary changes to the system (as it is often the case for system administrators of critical production servers). The solution that changes all the q_i to their most recent version might be preferred by "*trendy*" users willing to have a system as up to date as possible (which is the case for many desktop and developer users).

State of the art package managers try to handle this issue by incorporating hard-wired criteria (most of which would give preference to the trendy solution above) and sometimes provide a bit of flexibility by means of cumbersome mechanisms that let the user alter the standard solver behavior, like the *pinning* schema used by APT [28], or an API for programming custom criteria in Smart[2] and libzypp.[3]

Such ad-hoc mechanisms suffer from two main drawbacks: (1) they are package manager specific and therefore cannot be shared among different tools, preventing the development of common good practices in component deployment; (2) they are not expressive enough to encode all but the simplest use cases, making it difficult to precisely specify user needs. The right approach is—on the user side—to expose a high-level, solver independent, flexible mechanism to specify user preferences and—on the package manager side—to enable solver externalization and reuse.

3. Dependency solving as a separate concern

We have seen how dependency solving is a difficult, recurrent, and apparently underestimated problem. Re-developing from scratch dependency solvers as soon as dependencies and conflicts are introduced in yet another component model does not seem to have not served well users of component based systems. We argue that an alternative, more modular, approach is possible by treating *dependency solving as a separate concern* from other component management concerns. The goal is to decouple the evolution [sic] of dependency solving from that of specific package managers *and* component models.

We believe such a separation will benefit, at first, the involved scientific communities: CBSE and constraint solving. The former will gain the attention of the latter and will avoid to reinvent (solving) wheels, the latter will get access to a corpus of challenging upgrade problems to better tune existing solvers and techniques. Later on, we posit that synergies among the involved stakeholders will benefit final component users, by improving dependency solving abilities in state-of-the art package managers. Our early results seem to support these beliefs, as shown in Section 6.

To treat dependency solving as a separate concern, however, we need suitable abstractions and technologies that allow to describe upgrade problems in a way which is agnostic from specific component models and tools. In particular, we need ways to grasp all the information that describe any given upgrade problem instance:

1. *installed and available components*—describing all known components (local and remote) and information about which are currently installed;
2. *user request*—detailing the components that are requested to be installed, removed or upgraded, possibly with version constraints;
3. *user preferences*—the criteria describing how a user wants to choose a preferred solution out of the many possible ones.

In the following we present a Domain Specific Language (DSL)—called CUDF—able to encode (1.) and (2.), as well as a formalism defined on top of it to grasp (3.). Taken together they provide an unified way to capture all of the above in a unified way, which is both independent from component model details and rigorously defined to enable independent implementations of upgrade problem solvers. Having those devices available,

[2]`http://labix.org/smart`, retrieved December 2010
[3]`http://en.opensuse.org/Portal:Libzypp`, retrieved December 2010

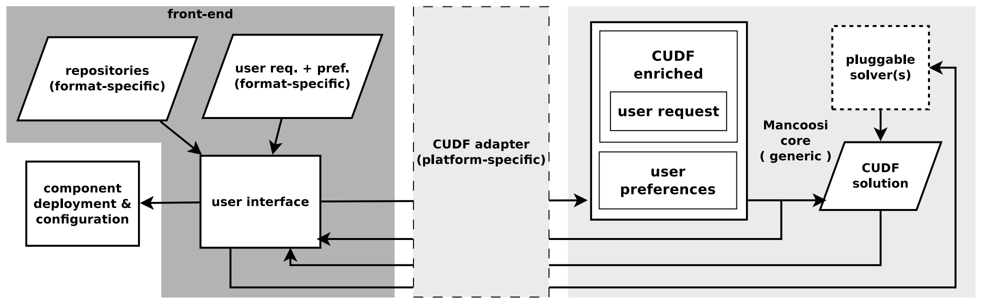
front-end
repositories (format-specific)
user req. + pref. (format-specific)
component deployment & configuration
user interface
CUDF adapter (platform-specific)
CUDF enriched
user request
user preferences
Mancoosi core (generic)
pluggable solver(s)
CUDF solution

Figure 3: Modular package manager architecture

we can build adapters for each component platform and then build a modular solver engine where solvers can be plugged in according to user needs. Even more so, solvers can be run in parallel locally or outsourced (e.g. to solver farms in the "cloud"), in order to provide the user with the best solution current techniques and technologies can find.

The resulting modular architecture is shown in Figure 3. In such an architecture separation of concerns is established as following: *package manager* developers may focus on the killer features of their software (trust management, user interface and interaction, transactional upgrade deployment, etc.) and stop worrying about dependency solving issues; *CUDF adapters* are created for each component model and maintained by component metadata architects, or by CUDF experts working with them; *dependency solvers* are maintained by solver experts, who will see their technology gain many new fields of application by just supporting one generic I/O format—CUDF—which comes with a rigorous semantics, relieving the pain of interpreting the meaning of platform-specific component metadata.

4. A unified description of upgrades

To enable treating dependency solving as a separate concern in component upgrade planning, we need a language able to capture all relevant aspects of upgrade problem instances. In this section we present a DSL called CUDF (for Common Upgrade Description Format), whose documents describe instances of the component upgrade problem. The design of CUDF has been guided by a few general principles:

Platform independence. CUDF is a *common* format to describe upgrade scenarios coming from diverse environments. As a consequence, CUDF makes no assumptions on specific component model, version schema, dependency formalism, or package manager.

Solver independence. In contrast to encodings of inter-component relations which are targeted at specific solver techniques (see Section 7), CUDF stays close to the original problem, in order to preserve its structure and avoid bias towards specific solver.

Readability. CUDF is a compact plain text format which makes it easy for humans to read upgrade scenario, and ease interoperability with package managers.[4]

[4]As evidence of the benefits of this choice, CUDF is routinely used by the Eclipse P2 team to reason about upgrade scenarios, instead of the native XML encoding that comes with Eclipse. See http:

Extensibility. Only core component properties that are shared by mainstream platforms and essential to grasp the meaning of upgrade scenarios are predefined in CUDF. Other auxiliary properties can be declared and used in CUDF documents, to allow the preservation of relevant information that can then be used in optimization criteria, e.g. component size, number of bugs, etc.
Formal semantics. CUDF comes with a rigorous semantics that allows package manager and solver developers to agree on the meaning of upgrade scenarios. For example, the fact that self-conflicts are ignored is not a tacit convention implemented by some obscure line of code, but a property of the formal semantics.

4.1. Language overview

An upgrade scenario is represented by a *CUDF document*. It consists of a sequence of *stanzas*, each of which is a collection of key-value pairs called *properties*. Properties are typed within a simple *type system* containing basic data types (integers, booleans, strings) and more complex, component-specific data types such as boolean formulae over versioned components used to represent inter-component relationships.

Each CUDF document is made up of three logical sections: a *preamble*, a component *universe*, and a *request*. The universe contains one *component stanza* for each component known to the package manager, so both installed and non-installed (but available) components are represented uniformly in a document, in contrast to current platforms which often distribute this information in different locations using different formats. Component stanzas support a set of core properties (possibly optional, with default values), the most important of which are: `package` and `version` (which uniquely identify a component in the universe), `depends` and `conflicts` (context requirements), `provides` (*features* provided by the component), and `installed` (whether the component is installed).

Figure 4 shows a sample CUDF document. The component universe contains several component stanzas, where both core and extra properties are used. Extra properties must be declared in the *preamble*, which starts the document. Extra properties account for extensibility of the format and enable type checking of CUDF documents. A *request stanza* encodes the user request and concludes the document. In its general form, the request stanza details the components the user wants to `install`/`remove`/`upgrade` (using the homonymous properties), possibly specifying version requirements.

The full syntax of CUDF is given, as an EBNF grammar, in Appendix A; the formal semantics in Appendix B.

4.2. Expressiveness

As CUDF lays at the "interface" between package managers and dependency solvers, its expressiveness should be validated looking from both angles. From the point of view of package managers, we have shown that upgrade scenarios coming from several major component models can be encoded in CUDF; adapters are already available for: Debian and RPM packages,[5] Eclipse [5]—with an extension for full OSGi bundles in the working—, and common feature diagram formalisms used in software product lines [11].

`//wiki.eclipse.org/Equinox/p2/Meetings/20091221`, retrieved November 2010.
[5]both are supported out-of-the-box by Mancoosi tools [24]

```
preamble:
property: bugs: int = 0, suite: enum(stable,unstable) = "stable",

package: car
version: 1
depends: engine, wheel > 2, door, battery <= 13
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true
...
request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 3
```

Figure 4: Sample CUDF document

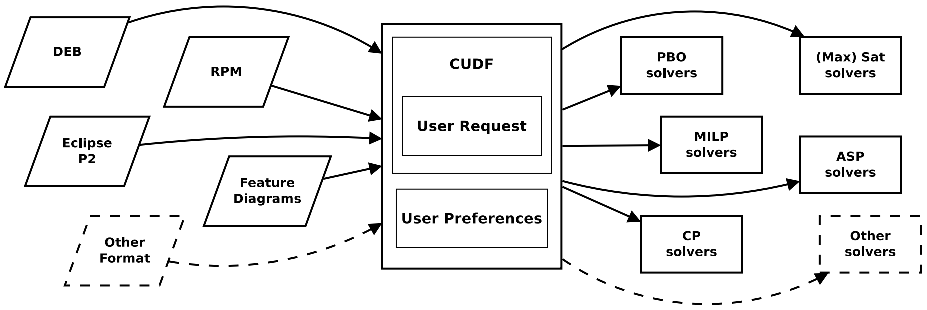

Figure 5: Sharing upgrade problems and solvers among communities

All encodings are *linear* in the number of components to encode, even in the presence of XOR dependencies.[6]

From the converse angle, that of dependency solvers, we observe that entrants in the MISC competition (see Section 6) have used very different solver technologies: boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Answer Set Programming (ASP), and graph constraints. They have all been able to handle upgrade problems encoded as CUDF documents, providing convincing evidence that CUDF is adequate for a large spectrum of solving techniques.

Hence, at the time of writing, CUDF is already a unique pivot format that allows on one hand to share solvers among different package managers, and on the other hand to share a corpus of challenging upgrade problems among solver communities, as shown in Figure 5. The number of supported solver technologies and component frameworks has

[6]while usual SAT encodings blow up quadratically in the number of XOR dependencies.

grown steadily over the past years and it is likely to keep growing in the future.

4.3. Implementations

CUDF has been subject to an ad-hoc standardization process, resulting in a specification [37]. `libcudf` is the "reference" implementation of the specification; it consists of a parsing and pretty-printing library for CUDF, as well as an implementation of CUDF semantics. The latter consists in:

1. given a CUDF document, `libcudf` can verify whether installed components are consistent, i.e. whether they satisfy abundance and peace;
2. given a CUDF document and an encoding of a potential solution, `libcudf` can verify whether the solution is valid, i.e. abundance, peace, and request satisfaction.

`libcudf` comes with the `cudf-check` command line tool which provides the above two features out of the box. `libcudf` is Free Software and can be used both from the OCaml and C programming languages; it is available at `http://www.mancoosi.org/software/`.

The authors are aware of other CUDF implementations. Some have been developed in the context of the Mancoosi project to capture FOSS distribution upgrade scenario descriptions into CUDF, in order to build a cross-distribution corpus of upgrade problem instances [3]. Using the tools we have verified that the average size of an upgrade scenario encoded in CUDF is linear with the size of the original package manager information and usually smaller, since metadata not relevant for describing the upgrade problem can be dropped. For instance, on a large Debian installation, using both testing and unstable suites (totaling $\approx$ 45'000 packages), APT information on disk amounts to 14 Mb while the corresponding CUDF document is only 9 Mb.

An independent CUDF implementation is also available in CUPT,[7] a recent APT-compatible package manager for Debian. In CUPT, CUDF is used as an interface format to pipe upgrade scenarios to external solvers, so that upgrade planning can be decoupled from other package manager activities. While no stable software has been released yet, work is ongoing to implement CUDF in APT and APT2 in order to decouple dependency solving from the package managers.

5. User preferences as multicriteria optimization

The DSL presented in the previous section addresses the need of grasping those aspects of an upgrade scenario that are related to the *correctness* of a given solution (i.e. "does the solution satisfy the user request as well as the expectations of all installed components?"). *Quality* aspects of solutions (i.e. "is the proposed solution to my liking?") are much less known, not to mention agreed upon, and hence they do not yet constitute suitable material for DSL standardization. Nonetheless, to improve the state-of-the-art in upgrade planning we do need at the very least a rigorous framework to reason about solution quality. In this section we propose one such formalism.

As we have seen in Section 2, there are in general exponentially many solutions to a user request, so it is necessary to allow users to express their preferences about the

[7] `http://wiki.debian.org/Cupt`, retrieved December 2010

desired solution. The state-of-the-art approach is to present one particular solution—found according to some built-in strategies generally unknown to the user—and then allow the user to interactively fiddle with the solution. Considering again Example 1, it is easy to see why this approach has serious shortcomings: a "paranoid" user who is presented with a "trendy" solution will need to make n changes to the solution (and usually rerun the solver each time) before getting what she wants. In modern component repositories, where n can be quite large, this approach is not viable.

An alternative approach is to let the user specify high-level criteria that capture what she considers important in a solution: she may be concerned about the packages that are *changed*, the packages that are *not up to date*, the packages that get *removed*, or even "the number of installed security fixes", or "the overall installed size". On top of CUDF semantics, we can build an extensible dictionary of well-defined criteria like the above and then let the user inform the solvers that the required solution should maximize, or minimize, a given criterion.

It is quite natural for the user to combine several of these criteria: to compare two solutions s and s' whose criteria have values $(c_1, \ldots, c_n)$ and $(c'_1, \ldots, c'_n)$, the user will prefer s over s' if all criteria of s are better or equal than s' (i.e. s is Pareto-better then s'). Unfortunately, when one has more than one criterion, there may be many incomparable Pareto-optimal solutions; this is the core problem of *multicriteria optimization* which has been extensively studied in the optimization research community [34]. Many different approaches have been proposed to aggregate multiple criteria, the most common being:

Lexicographic. The criteria are ordered by importance, and compared lexicographically: $(c_1, \ldots, c_n)$ is better than $(c'_1, \ldots, c'_n)$ iff there exists a i s.t. for all $j < i$ $c_j = c'_j$, and $c_i > c'_i$; for example, a security upgrade may be considered more important than any other criterion, and put first in the order.

Weighted sum. The criteria are aggregated into a single measure using user-specified *weights* k_i: $(c_1, \ldots, c_n)$ is better than $(c'_1, \ldots, c'_n)$ iff $\sum_{1 \leq i \leq n} k_i c_i > \sum_{1 \leq i \leq n} k_i c'_i$; this may be useful when trying to balance different criteria for which no clear order is established.

More sophisticated approaches exists, like *leximin* and *leximax* [13], and an extensive literature is devoted to them. According to the use case, the best aggregation function may vary widely. Our own proposal for a high level user preferences formalism is simple yet expressive:

1. define a dictionary of useful criteria c_i;
2. define a dictionary of aggregation functions *lex*, *weightedsum*, *leximin*, etc.
3. write the user preference as an expression $op(k_1 c_1, \ldots, k_n c_n)$ where k_i can be one of $\{+, -\}$ to indicate maximization or minimization of the criterion (for aggregation functions like *lex*, *leximin* and *leximax*), or an integer (for aggregation functions like *weightedsum*).

Formally we define the criteria as in Table 1, where I is the initial installation and S is a proposed new installation. We write $V(X, name)$ the set of versions in which *name* (the name of a component) is installed in X, where X may be I or S. That set may be empty (*name* is not installed), contain one element (*name* is installed in exactly that version), or even contain multiple elements in case a component is installed in multiple

Table 1: Optimization criteria

$removed(I, S)$	$=\{name \mid V(I, name) \neq \emptyset$ and $V(S, name) = \emptyset\}$
$new(I, S)$	$=\{name \mid V(I, name) = \emptyset$ and $V(S, name) \neq \emptyset\}$
$changed(I, S)$	$=\{name \mid V(I, name) \neq V(S, name)\}$
$notuptodate(I, S)$	$=\{name \mid V(S, name) \neq \emptyset$ and does not contain the most recent version of name in $S\}$
$unsatrec(I, S)$	$=\{(name, v, c)$ — v is an element of $V(S, name)$ and $(name, v)$ recommends ..., c, ... and c is not satisfied by $S\}$

versions.[8] Using this formalism, it is quite easy to define a *paranoid* preference as

$$paranoid = lex(-removed, -changed)$$

The solution scoring best under this criterion will be the one with the minimum number of removed functionalities, and then with the minimum number of changes. A *trendy* preference is also easy to write

$$trendy = lex(-removed, -notuptodate, -unsatrec, -new)$$

Currently, each criterion and aggregation function must be specifically encoded for a given solver technology, but work on a generic system which will be able to produce these encodings automatically is ongoing [38].

6. Experimental results: the Mancoosi International Solver Competition

The DSL and formalism presented in the previous two sections have been used to run a dependency solving competition called MISC, for Mancoosi International Solver Competition. The variety of solving techniques implemented by participants, as well as the popularity of FOSS distributions from which package manager entrants come, give, in the authors opinion, a reasonable guarantee of the generality of the following results, that come from MISC 2010, the first edition of the competition:

1. The proposed languages and formalisms are expressive enough to encode both *real* upgrade scenarios coming from users of popular FOSS distributions and *synthetic* problems of increasing complexity.
2. The proposed languages and formalisms are unbiased enough to allow constraint solvers, based on a wide range of techniques, to attack upgrade problem instances. Dependency solving can therefore be outsourced to external solvers, as depicted in Figure 3.
3. The complexity of real upgrade problem instances grows with the number of component repositories, as well as the complexity of the optimization criteria.

[8]The CUDF component model is not flat but allows to encode both flat and non-flat models [3].

4. Dependency solving abilities of package managers used in popular FOSS distributions fall short of state-of-the-art constraint solvers, both in terms of solution quality and completeness.

In this section we present and discuss MISC 2010 results, as evidence of the above claims.

6.1. Competition details

MISC 2010 has been run in June 2010. Its results have been presented at the LoCoCo workshop, in the context of FLoC (Federated Logic Conference) 2010. All competition data (formal rules, problem instances, results, etc.) are available at `http://www.mancoosi.org/misc-2010/` and allow to independently re-run the competition.

Each participant had to face several problem instances. For each instance, the solver received a full CUDF document as input and must produce a CUDF-encoded solution (i.e. a CUDF document without a request stanza). Solvers could participate in either one or both of two tracks—trendy and paranoid, as defined in Section 5—and strove to optimize their solutions accordingly. Problem instances were classified in categories: synthetic problems (categories: *easy*, *difficult*, *impossible*), instances of the problem 1-in-3 SAT (category *cudf_set*), and real instances collected from Debian users (category *debian-dudf*) using the `mancoosi-contest` utility [24] which plugs into package managers for Debian-based distributions to store upgrade problems in CUDF format.

Synthetic problems have been generated from a real Debian installation by varying a number of parameters such as the number of components in the universe, the number of installed components, and the number of components requested to install/remove/upgrade (i.e. request size). The size of requests ranges from 10 (for *easy*) to 20 (*impossible*) and components appearing therein are possibly equipped with version constraints.

For the *difficult* and *impossible* categories, the initial state has been made on purpose inconsistent by marking random components as installed, ensuring their context requirements were not satisfied, to simulate a badly broken user installation.

The following solvers took part in the competition:

solver	**author/affiliation**	**technique/solver**
apt-pbo [39]	Trezentos / Caixa Magica	Pseudo Boolean Optimization
aspcud	Matheis / University of Potsdam	Answer Set Programming
inesc [4]	Lynce et. al / INESC-ID	Max-SAT
p2cudf [4]	Le Berre and Rapicault / Univ. Artois	Pseudo Boolean Optimization / Sat4j (`www.sat4j.org`)
unsa [26]	Michel et. al / Univ. Sophia-Antipolis	Mixed Integer Linear Programming / CPLEX (`www.cplex.com`)

No solver has been provided by the authors, who acted solely as competition organizers. We added two extra participants—apt-get and aptitude—by wrapping with a CUDF-compatibility layer the solvers of package managers used in Debian-based FOSS distributions. As they do not allow to specify preferences, the purpose of the experiment was to check how hard-coded optimizations score with respect to competition criteria. The solver *ucl* from Gutierrez et. al from Univ. Louvain that took part to the Misc competition it is not presented here as its results were not relevant.

Table 2: MISC 2010 results: paranoid (above) and trendy (below) criteria. Each column show (before the parenthesis) the penalties accumulated by a participant, category by category: the lower, the better. Between parentheses is shown the aggregate solving time, in seconds, of a participant on all the problems of a given category: the lower the better, although time is used only to break ties between participants who accumulated the same amount of penalties. Highlighted cells, one per row, denote the winning participant of a given category.

Category	apt-pbo	aspcud	inescp	p2cudf	uns
cudf_set	162 (2700.00)	118 (1572.82)	108 (11.64)	108 (15.22)	111 (1984.21)
debian-dudf	236 (3673.76)	222 (3700.45)	32 (271.39)	32 (180.01)	86 (1336.49)
difficult	522 (1039.71)	58 (1553.82)	92 (298.99)	99 (3209.55)	55 (34.97)
easy	504 (177.10)	21 (408.48)	63 (122.44)	63 (121.97)	21 (18.59)
impossible	300 (3873.00)	270 (4500.00)	120 (1890.13)	120 (1924.79)	15 (151.27)
Total	1724 (11463.58)	689 (11735.57)	415 (2594.58)	422 (5451.54)	288 (3525.52)

Category	apt-pbo	aspcud	inesct	p2cudf	uns
cudf_set	135 (2700.00)	93 (2007.25)	90 (12.38)	90 (17.97)	107 (2332.43)
debian-dudf	211 (3743.04)	189 (3881.19)	194 (3711.60)	39 (5151.68)	74 (1782.70)
difficult	415 (1959.87)	70 (4439.18)	101 (1681.03)	98 (5223.88)	49 (83.80)
easy	410 (222.92)	46 (1034.32)	64 (317.52)	61 (597.80)	21 (24.85)
impossible	225 (4500.00)	190 (2671.69)	220 (4221.98)	181 (4182.90)	15 (734.58)
Total	1396 (13125.83)	588 (14033.63)	669 (9944.52)	469 (15174.24)	266 (4958.36)

Table 3: Solver comparison with a growing number of component repositories; paranoid (above) and trendy (below) criteria. Categories are identified by the initial(s) of Debian repositories (or *suites*), in the following order: `sarge`, `etch`, `lenny`, `squeeze`, `sid`. Cell values are to be interpreted in the same way as Table 2.

Category	apt-get	aptitude	aspcud	inescp	p2cudf	uns
s-e-l-s-s	176 (2435.49)	252 (1672.11)	210 (3000.00)	77 (1070.93)	24 (2861.72)	10 (78.13)
s-e-l-s	176 (2434.13)	231 (2331.87)	210 (3000.00)	58 (818.82)	23 (2855.84)	10 (63.90)
s-e-l	210 (3000.00)	245 (1621.60)	10 (389.68)	48 (443.98)	32 (1757.80)	10 (30.23)
s-e	194 (2706.06)	280 (116.07)	10 (166.57)	30 (84.17)	30 (555.35)	10 (14.64)
s	82 (918.40)	57 (33.80)	10 (68.14)	47 (31.87)	47 (268.22)	10 (3.68)
Total	838 (11494.07)	1065 (5775.45)	450 (6624.40)	260 (2449.77)	156 (8298.92)	50 (190.60)

Category	apt-get	aptitude	aspcud	inesct	p2cudf	uns
s-e-l-s-s	150 (2435.50)	198 (2419.24)	174 (2731.37)	180 (3000.00)	36 (2882.61)	10 (401.82)
s-e-l-s	150 (2433.72)	192 (2617.81)	180 (3000.00)	180 (3000.00)	20 (2866.66)	10 (318.83)
s-e-l	180 (3000.00)	204 (2000.62)	20 (2013.05)	50 (1949.77)	34 (2854.06)	10 (121.50)
s-e	166 (2706.06)	240 (208.88)	112 (2700.21)	26 (422.95)	31 (2281.39)	10 (34.35)
s	88 (918.37)	61 (34.55)	92 (2259.55)	34 (57.33)	34 (1158.47)	10 (5.32)
Total	734 (11493.65)	895 (7281.09)	578 (12704.17)	470 (8430.05)	155 (12043.18)	50 (881.82)

MISC 2010 results are given in Table 2. The clear winner in both tracks is *unsa*, followed by *p2cudf* for the trendy track and *inesc* for the paranoid track. It is important to notice that the solvers perform differently on different problem sets: for example, *p2cudf* shows better results than the others in the category *debian-dudf*, and it will be surely interesting to analyse, in future work, the structural differences among the different problem sets.

6.2. Discussion

CUDF acceptance. The actual run has been preceded by a discussion period among organizers and participants. During this period, solver authors could expose their doubts about CUDF semantics and competition rules, as well as submit solver prototypes for the only purpose of testing their CUDF-based interface with the competition infrastructure.

Solver authors have not reported any perceived bias, of either CUDF or the optimization criteria, towards specific solving techniques. The acceptance of the proposed languages and formalisms among participants has hence been very good, although self-selection bias is possible. The main discussion topics revolved around parsing issues and misconceptions about how upgrades "should" work. Interestingly, while CUDF semantics is rigorous and has proven to be very stable thus far, solver authors used to specific component platforms tend to believe upgrade should work as they are "used to", even if the semantics of upgrades in their platform of origin (e.g. RPM) is ambiguous and delegated to implementation details of specific package managers. This aspect has reinforced our conviction that an interface format equipped with a rigorous semantics is the way to go in order to drive the attention of constraint solving communities to upgrade problem issues.

How complexity grows in practice. MISC 2010 results clearly show that the number of criteria in the optimization function is an important source of complexity: the trendy and paranoid tracks are run on the same problem sets, whereas trendy (which has more parameters than paranoid) is consistently more difficult to handle for all solvers.

We have also run all competition entrants on a separate set of problems, specifically designed to test the impact of having several repositories available, a scenario that happens quite often in practice. The corresponding categories have been built as follows: in a base universe, corresponding to the Debian distribution *sarge* (currently also known as *oldstable*), a fixed request and a fixed set of installed components is generated, ensuring it is satisfiable (meaning that the request has at least a solution, independently of the optimization criteria). The very same upgrade problem is then replicated in larger universes, by adding more recent Debian repositories: *etch*,*lenny*, *squeeze*, and *sid*.

Table 3 show the performance of the solvers on these categories. It is immediate to notice how the time needed to answer the same request grows very quickly when increasing the number of available component repositories. This is explained by the fact that using multiple repositories greatly increases the number of components available in multiple versions and, in turn, the number of conflicts in a flat component model.

Performance of FOSS package managers. Table 3 permits to assess the relative performances of competition entrants and package managers from popular FOSS distributions. Package manager solvers exhibit decent performances on machines equipped with a single component repository, which is often the case for freshly installed machines.

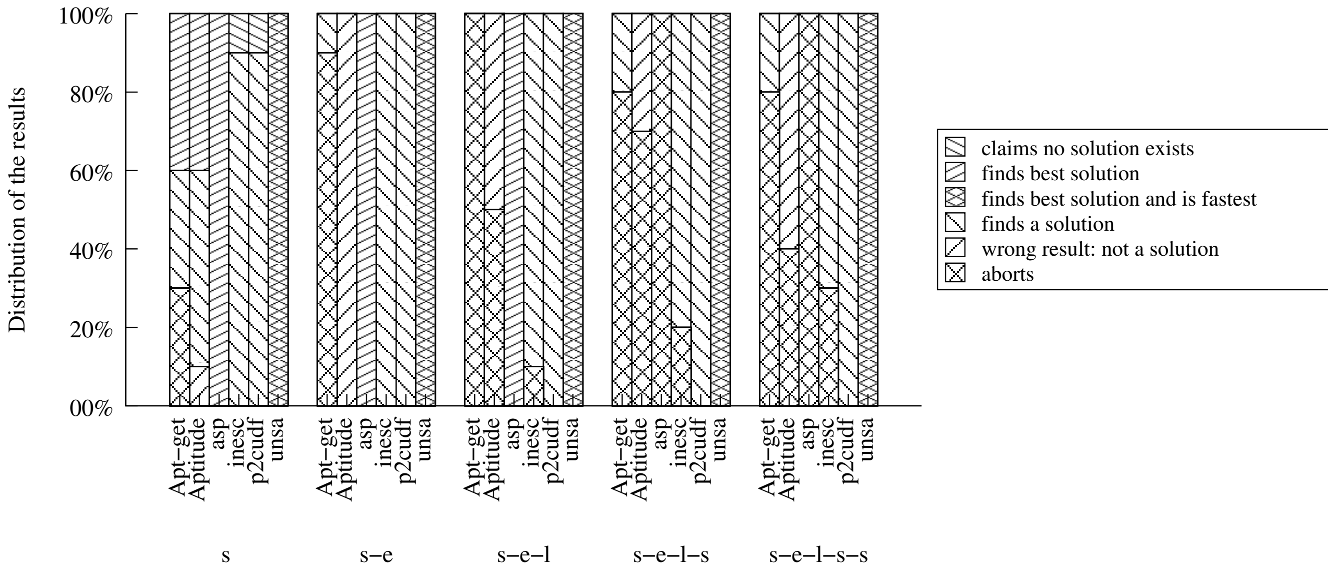

Figure 6: Solver results with increasing number of repositories (trendy)

This matches end-user experience that package manager performance on newly installed machines is quite good.

However, problems on machines that were installed from an old distribution and that use a mixture of component repositories, turn out to be very challenging. In Figure 6 we see clearly that, while *apt-get* and *aptitude* behave well with one repository, they become unreliable from two repositories on, and are no longer able to solve the large majority of problems at all. This corresponds to the end-user experience that installation and upgrades become less reliable on FOSS machines after a year or so: this corresponds more or less to the release cycle of several mainstream distributions, and end-users find themselves on machines where the package manager needs to handle more than one repository, the original one from which the machine was installed, and the newly released one.

Looking at the time distribution of the same experimental data in Figure 7 (for the trendy criterion) we observe a similar pattern. While state-of-the-art package managers abort or time-out, solvers like *unsa* or *inesc* are still able to cope with complex problems in less than 60 seconds. It is important to notice that in this context, a solution is "optimal" only with respect to solutions given by other solvers. Therefore even if a solver does not provide the best solution, it is still important to take it into consideration in the overall evaluation.

The data for the paranoid criterion in Table 3 shows the same overall behaviour, with significantly shorter execution times, indicating that the number of combined criteria in the user preferences is another significant factor in the complexity of the upgrade problems (paranoid involves 2 criteria, while trendy involves 4).

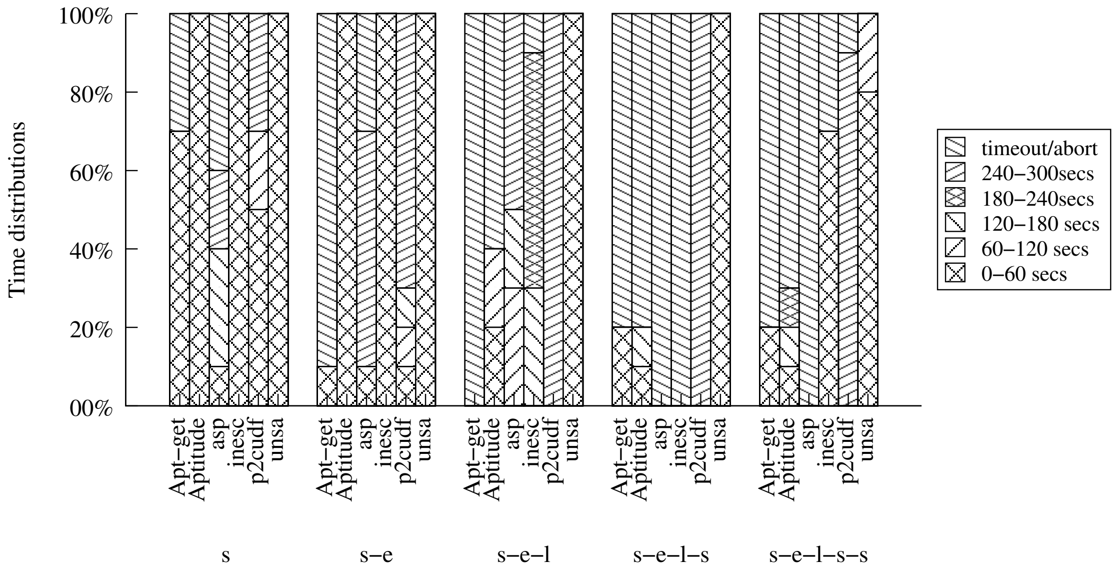

Figure 7: Solver performances with increasing number of repositories (trendy)

7. Related work

Software evolution management has many facets; in this paper we have focused on the area of post-development management. In particular we have studied how to improve software upgrade planning in package managers which are equipped with automatic dependency solving, given that such utilities are common place in component-intensive software platforms. The problem of dealing with inter-component relationships was known well-before the advent of such package managers, though. Seminal work in the area of software configuration management (e.g. [8, 27]) has established the “provide/require” paradigm to reason about component interconnection, with a varying degree of granularity and expressiveness [30]. Those and subsequent works have also detailed formal properties able to grasp, and practically verify, the compatibility of (new versions of) components within a given deployment context.

The explicit notion of inter-component conflicts is not part of those seminal proposals. In the technology camp such a notion has been popularised by the advent of early component managers (e.g. the FreeBSD porting system [35], RPM, and dpkg). Together with conflicts, early package managers have brought to users the folklore problem known as “dependency hell”, i.e. the difficulty of satisfying at the same time all component dependencies and conflicts, sometimes stumbling upon the (apparent) impossibility of doing so. The next technology leap has brought package managers equipped with automatic dependency solving abilities (e.g. APT [28], Yum, Urpmi, etc.). Such systems have not only solved most of the issues brought by the dependency hell, but also addressed several of the concerns related to software distribution (see [16] for an introduction on the subject), even though they have done so in a centralized rather than federated way [15]. Where the state of the art in package managers is still lacking though, as we have shown, is in their actual dependency solving ability. Improvement in that area is badly needed to properly plan upgrades in component-intensive software platforms.

Turning to formal encodings of the upgrade problem, an early SAT-encoding and complexity analysis for the upgrade problem, limited to the component models of FOSS distributions, has been provided by some of the authors in [12, 23] and has popularised the use of SAT technology in package managers. Results and encoding detailed in the present paper are more general and detail the minimum requirements for any component model to exhibit similar complexity behaviors.

The OPIUM prototype used in 2006 a SAT solver with an ad-hoc, hard coded optimization in line with the paranoid criterion [40]; SUSE's libzypp incorporated a SAT solver in 2007; the Eclipse P2 system comes with the Sat4J solver since 2007 [18]. This trend seems to continue steadily: a very recent entrant is apt-pbo, introduced in the Caixa Mágica distribution in early 2010 [39]. None of those systems have offered the ability to outsource dependency solving and optimization to an external solver.

The language we have proposed to encode user preferences is more flexible than those of OPIUM and similar experiences: we provide a core ontology of criteria and combinators to join them together. Even though user criteria must currently be specifically encoded for any given solver, we have looked at ways to automate the encoding. Moreover instead of leaving to the user the task of defining specific criteria, they could be asked for high level preferences and then use a goal-based model to "compile" those desiderata to the target criterion language. The work of Liaskos et al. [22], even if not directly related to our domain, goes in the direction of making complex systems easily configurable by deducing low-level options from high-level user specifications.

Several alternative encodings of the upgrade problem have been proposed: SAT [23, 40, 18], Pseudo Boolean Optimization [39], Partial Weighted Max SAT [4], Mixed Integer Linear Programming [26], as well as some others championed by entrants in the MISC 2010 competition (see Section 6).

Jenson [17] proposes a component model without explicit (or implicit) component conflicts and does not handle component removal in neither requests nor solutions. As a consequence, such a degenerate upgrade problem is way simpler than what we have modeled in this paper and can be solved in polynomial time, even though the number of solutions may be huge. Dependency solving as SAT with optimization has been reviewed in [18] where it was also observed that much of the complexity stems from multiple versions of components and the constraints they entail.

The need of dealing properly with dependencies in CBSE have been observed before [21, 41]. Vieira et. al have argued that dependencies should be treated as a first class problem in CBSE [41] and have established requirements for that. While we focus on static deploy-time dependencies, which have become popular in the meantime, we observe that CUDF fulfills all their requirements of "*being based on uniform design principles following some kind of standardization*" and offer dependency metadata which are "*expressive, intuitive, and concise* [in] *representation*". We agree with the authors and believe that the proposed formalisms are a significant step forward in treating dependencies as a first class problem in CBSE.

8. Conclusions

Dependency solving is difficult. This is hardly a surprise for anyone maintaining software installations, especially when they are made of thousands of components evolving rapidly and independently. The phenomenon requires nevertheless a detailed analysis to pinpoint the origin of the complexity. We found that, for common component models and platforms, the complexity is due to inter-component conflicts, either explicitly declared as component metadata or implicitly assumed between different versions of the same components. This theoretical result is confirmed by experimentation on both real and synthetic upgrade problem instances: dependency solving becomes harder as component repositories are added, thus increasing the number of available versions of the same components. Complexity also increases with the complexity of user preferences (i.e. optimization criteria). This explains why shortcomings of state-of-the-art dependency solvers are often not observed on freshly installed machines, but pop up as soon as one tries to do upgrades among distribution major releases, or else to mix and match components from different releases.

Better tools to support evolution of component-based systems are needed. Design, development, integration, and deployment of these new tools will only be made possible if we treat dependency solving as a separate concern of evolution management, i.e. as a first class research problem in its own right. To that end, we need rigorous abstractions to be put at the interface between component managers and solvers engineered by independent research communities, which enjoy the challenges posed by concrete upgrade problems. We have introduced some of those abstractions—CUDF and a companion user preference language—and we have reported the results of MISC, an international solver competition based on these abstraction, which confirm their adequateness.

On top of the proposed abstractions, it is easy to imagine a generic component manager front-end, which implements the architecture of Figure 3 and can then be targeted, adding back-ends, to specific component platforms. We have developed one such prototype, called MPM [2], targeting Debian-based FOSS distributions. Any solver implementing the interface of MISC 2010 can be plugged into MPM and used to plan package upgrades; upgrade deployment will then be delegated to legacy distribution tools. As an example, a trivial solution to the upgrade problem discussed in Figure 2 can be found by MPM using the *inesc* solver submitted to the paranoid track:

```
remove: gnome-utils gnome-desktop-environment gnome
install: baobab=2.4.2-1.1+b1
```

in such a solution the (virtual) packages `gnome-utils`, `gnome-desktop-environment`, and `gnome` are still removed, whereas all other packages forming the GNOME desktop are not, saving the (possibly newbie) user from losing her user-friendly work environment. This is a consequence of the paranoid criterion and of a dependency solver able to find a high-quality solution with respect to such desiderata.

As this example shows, one-size–fits-all solvers are not the way to go, especially when solvers are developed in house without reusing existing knowledge and results. Rather, we need highly customizable upgrade planners able to satisfy diverse user needs. Decoupling solvers from package managers is a necessary intermediate step that makes it possible to experiment with independent solvers, and to outsource dependency solving to evolution planners living far away from component managers.

As evidence of the pertinence of our approach, the experimental version `0.8.16 exp5` of `apt`, a mainstream package manager for the Debian distribution, implements a CUDF interface to call the solvers issued from the MISC competition.

Acknowledgments

The authors are grateful to the members of the Mancoosi and Eclipse P2 projects, for many stimulating discussions on the CUDF format, optimization criteria, and their practical use cases. A special acknowledgement goes to the optimization and solving communities who took part in the editions of the MISC competition: their solvers are the ingredients that make separation of concerns around dependency solving a concrete and useful approach.

References

[1] Abate, P., Boender, J., Di Cosmo, R., Zacchiroli, S., 2009. Strong dependencies between software components, in: ESEM 2009: International Symposium on Empirical Software Engineering and Measurement, IEEE. pp. 89–99.

[2] Abate, P., di Cosmo, R., Treinen, R., Zacchiroli, S., 2011. Mpm: A modular package manager, in: CBSE 2011, ACM.

[3] Abate, P., Guerreiro, A., Laurière, S., Treinen, R., Zacchiroli, S., 2010. Extension of an existing package manager to produce traces of ugradeability problems in CUDF format. Mancoosi deliv. D5.2. `http://www.mancoosi.org/reports/d5.2.pdf`.

[4] Argelich, J., Le Berre, D., Lynce, I., Marques-Silva, J., Rapicault, P., 2010. Solving Linux upgradeability problems using boolean optimization, in: LoCoCo: Logics for Component Configuration, pp. 11–22.

[5] Bozman, C., 2010. Converting Eclipse metadata into CUDF. Technical report 5. Mancoosi project. `http://www.mancoosi.org/reports/p2.pdf`.

[6] Brown, A.W., Wallnau, K.C., 1998. The current state of CBSE. IEEE Software 15, 37–46.

[7] Cook, S., Harrison, R., Lehman, M.M., Wernick, P., 2006. Evolution in software systems: foundations of the SPE classification scheme. Journal of Software Maintenance and Evolution 18, 1–35.

[8] DeRemer, F., Kron, H., 1975. Programming-in-the large versus programming-in-the-small. SIGPLAN Notice 10, 114–121.

[9] Des Rivières, J., Wiegand, J., 2004. Eclipse: a platform for integrating development tools. IBM Systems 43, 371–383.

[10] Di Cosmo, R., Trezentos, P., Zacchiroli, S., 2008. Package upgrades in FOSS distributions: Details and challenges, in: HotSWUp'08: International Workshop on Hot Topics in Software Upgrades, ACM. pp. 7:1–7:5.

[11] Di Cosmo, R., Zacchiroli, S., 2010. Feature diagrams as package dependencies, in: SPLC 2010: Software Product Lines Conference, pp. 476–480.

[12] EDOS Project, 2006. Report on Formal Management of Software Dependencies. EDOS Project Deliverables D2.1 and D2.2.

[13] Fishburn, P.C., 1974. Lexicographic orders, utilities and decision rules: A survey. Management Science 20, 1442–1471.

[14] Gonzalez-Barahona, J., Robles, G., Michlmayr, M., Amor, J., German, D., 2009. Macro-level software evolution: a case study of a large software compilation. Empirical Software Engineering 14, 262–285.

[15] Hall, R.S., Heimbigner, D., van der Hoek, A., Wolf, A.L., 1997. An architecture for post-development configuration management in a wide-area network, in: 17th International Conference on Distributed Computing Systems, IEEE. pp. 269–278.

[16] van der Hoek, A., Hall, R., Heimbigner, D., Wolf, A., 1997. Software release management, in: Software Engineering — ESEC/FSE'97. Springer Berlin / Heidelberg. volume 1301 of *LNCS*, pp. 159–175.

[17] Jenson, G., Dietrich, J., Guesgen, H., 2010. An empirical study of the component dependency resolution search space, in: CBSE 2011: International ACM Sigsoft Symposium on Component Based Software Engineering, Springer. pp. 182–199.

[18] Le Berre, D., Parrain, A., 2008. On SAT technologies for dependency management and beyond, in: SPLC 2008: Software Product Lines Conference, 2nd Volume, pp. 197–200.

[19] Le Berre, D., Rapicault, P., 2009. Dependency management for the Eclipse ecosystem, in: IWOCE 2009: International Workshop on Open Component Ecosystems, ACM. pp. 21–30.

[20] Lehman, M., 1980. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68, 1060–1076.

[21] Lehman, M.M., Ramil, J.F., 2000. Software evolution in the age of component-based software engineering. IEEE Proceedings 147, 249–255.

[22] Liaskos, S., Lapouchnian, A., Wang, Y., Yu, Y., Easterbrook, S., 2005. Configuring common personal software: a requirements-driven approach, in: Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on, pp. 9 – 18.

[23] Mancinelli, F., Boender, J., Di Cosmo, R., Vouillon, J., Durak, B., Leroy, X., Treinen, R., 2006. Managing the complexity of large free and open source package-based software distributions, in: ASE 2006: Automated Software Engineering, IEEE. pp. 199–208.

[24] Mancoosi, 2010. Mancoosi software tools. http://www.mancoosi.org/software/. Retrieved December 2010.

[25] Massol, V., O'Brien, T.M., 2005. Maven: A Developer's Notebook. O'Reilly Media.

[26] Michel, C., Rueher, M., 2010. Handling software upgradeability problems with MILP solvers, in: LoCoCo 2010: Logics for Component Configuration, pp. 1–10.

[27] Narayanaswamy, K., Scacchi, W., 1987. Maintaining configurations of evolving software systems. IEEE Transactions on Software Engineering 13, 324–334.

[28] Noronha Silva, G., 2008. APT howto. http://www.debian.org/doc/manuals/apt-howto/.

[29] OSGi Alliance, 2003. OSGi Service Platform, Release 3. IOS Press, Inc.

[30] Perry, D.E., 1987. Software interconnection models, in: 9th International Conference on Software Engineering, IEEE Computer Society Press. pp. 61–69.

[31] Raymond, E.S., 2001. The cathedral and the bazaar. O'Reilly.

[32] Schaefer, T.J., 1978. The complexity of satisfiability problems, in: 10th Annual ACM Symposium on Theory of Computing, ACM. pp. 216–226.

[33] Schmid, K., 2010. Variability modeling for distributed development — a comparison with established practice, in: SPLC 2010: Software Product Lines Conference, Springer. pp. 151–165.

[34] Steuer, R.E., 1986. Multiple Criteria Optimization: Theory, Computation and Application. Wiley.

[35] Stokely, M., 2004. The FreeBSD Handbook. FreeBSD Mall. 3 edition.

[36] Szyperski, C., 1998. Component Software. Beyond Object-Oriented Programming. Addison-Wesley.

[37] Treinen, R., Zacchiroli, S., 2009a. Common Upgradeability Description Format (CUDF) 2.0. Technical Report 3. The Mancoosi Project. http://www.mancoosi.org/reports/tr3.pdf.

[38] Treinen, R., Zacchiroli, S., 2009b. Expressing advanced user preferences in component installation, in: IWOCE 2009: International Workshop on Open Component Ecosystems, ACM. pp. 31–40.

[39] Trezentos, P., Lynce, I., Oliveira, A., 2010. Apt-pbo: Solving the software dependency problem using pseudo-boolean optimization, in: ASE'10: Automated Software Engineering, ACM. pp. 427–436.

[40] Tucker, C., Shuffelton, D., Jhala, R., Lerner, S., 2007. OPIUM: Optimal package install/uninstall manager, in: ICSE'07: International Conference on Software Engineering, IEEE. pp. 178–188.

[41] Vieira, M., Richardson, D., 2002. The role of dependencies in component-based systems evolution, in: IWPSE'02: International Workshop on Principles of Software Evolution, ACM. pp. 62–65.

Appendix A. CUDF syntax

Overall structure.

cudf ::= *preamble*? *universe* *request*

Flow elements.

ssep ::= (*comment* | ‘`\n`’) ∗ ‘`\n`’ (*comment* | ‘`\n`’)∗
comment ::= ‘`#`’ *line*
line ::= `[^\n]` ∗ ‘`\n`’

Document parts.

preamble ::= ‘`preamble: `’ *line* *stanza* *ssep*
universe ::= *package*∗
package ::= ‘`package: `’ *pkgname* *stanza* *ssep*
request ::= ‘`request: `’ *line* *stanza* *comment*∗

Stanzas.

stanza ::= (*property* ‘`\n`’ | *comment*)∗
property ::= *propname* ‘`: `’ *value*
propname ::= *ident*
value ::= *bool* | *enum* | *int* | *nat* | *posint* | *string* | *pkgname* | *ident* | *typedecl*
| *vpkg* | *veqpkg* | *vpkgformula* | *vpkglist* | *veqpkglist*

Values: CUDF types.

bool ::= ‘`true`’ | ‘`false`’
int ::= (‘`+`’|‘`-`’)? `[0-9]`+
string ::= `[^\r\n]`∗
vpkg ::= *pkgname* (*sp* + *vconstr*)?
vpkgformula ::= *andfla* | ‘`true!`’ | ‘`false!`’
vpkglist ::= ‘’ | *vpkg* (*sp* ∗ ‘`,`’ *sp* ∗ *vpkg*)∗
enum ::= *ident*
pkgname ::= `[A-Za-z0-9+./@()%-]`+
ident ::= `[a-z][a-z0-9-]`∗
nat ::= ‘`+`’`[0-9]`+
posint ::= ‘`+`’`[0-9]` ∗ `[1-9][0-9]`∗
veqpkg ::= *pkgname* (*sp* + *veqconstr*)?
veqpkglist ::= ‘’ | *veqpkg* (*sp* ∗ ‘`,`’ *sp* ∗ *veqpkg*)∗
typedecl ::= ‘’ | *typedecl1* (*sp* ∗ ‘`,`’ *sp* ∗ *typedecl1*)∗

Value: gory details.

$$\begin{aligned}
\textit{vconstr} &::= \textit{reop}\ \textit{sp}+\ \textit{ver}\\
\textit{veqconstr} &::= \text{`}\texttt{=}\text{'}\ \textit{sp}+\ \textit{ver}\\
\textit{relop} &::= \text{`}\texttt{=}\text{'} \mid \text{`}\texttt{!=}\text{'} \mid \text{`}\texttt{>=}\text{'} \mid \text{`}\texttt{>}\text{'} \mid \text{`}\texttt{<=}\text{'} \mid \text{`}\texttt{<}\text{'}\\
\textit{sp} &::= \text{`}\ \text{'} \mid \text{`}\texttt{\textbackslash t}\text{'}\\
\textit{ver} &::= \textit{posint}\\
\textit{andfla} &::= \textit{orfla}\ (\textit{sp}*\ \text{`}\texttt{,}\text{'}\ \textit{sp}*\ \textit{orfla})*\\
\textit{orfla} &::= \textit{atomfla}\ (\textit{sp}*\ \text{`}\texttt{|}\text{'}\ \textit{sp}*\ \textit{atomfla})*\\
\textit{atomfla} &::= \textit{vpkg}\\
\textit{typedecl1} &::= \textit{ident}\ \textit{sp}*\ \text{`}\texttt{:}\text{'}\ \textit{sp}*\ \textit{typeexpr}(\textit{sp}* = \textit{sp}*\ \text{`}\texttt{[}\text{'}\ \textit{value}*\ \text{`}\texttt{]}\text{'})?\\
\textit{typeexpr} &::= \textit{typename} \mid \text{`}\texttt{enum}\text{'}\ \textit{sp}*\ \text{`}\texttt{[}\text{'}\ \textit{ident}\ (\text{`}\texttt{,}\text{'}\ \textit{sp}*\ \textit{ident})*\ \text{`}\texttt{]}\text{'}\\
\textit{typename} &::= \text{`}\texttt{bool}\text{'} \mid \text{`}\texttt{int}\text{'} \mid \text{`}\texttt{nat}\text{'} \mid \text{`}\texttt{posint}\text{'} \mid \text{`}\texttt{string}\text{'} \mid \text{`}\texttt{pkgname}\text{'}\\
&\quad \mid \text{`}\texttt{ident}\text{'} \mid \text{`}\texttt{vpkg}\text{'} \mid \text{`}\texttt{veqpkg}\text{'} \mid \text{`}\texttt{vpkgformula}\text{'} \mid \text{`}\texttt{vpkglist}\text{'}\\
&\quad \mid \text{`}\texttt{veqpkglist}\text{'}
\end{aligned}$$

Appendix B. CUDF semantics

Appendix B.1. CUDF types

We start by defining the domains of CUDF types, which are used in the definition of the semantics later on.

Definition 1 (CUDF type domains)**.**

- $\mathcal{V}(\texttt{posint})$ *is the set of* positive *natural numbers*
- $\mathcal{V}(\texttt{ident})$ *is a set of distinguished labels (intuitively, there is one such label for each lexically valid CUDF identifier)*
- $\mathcal{V}(\texttt{bool})$ *is the set* $\{\texttt{true}, \texttt{false}\}$
- $\mathcal{V}(\texttt{vpkgformula})$ *is the smallest set F such that:*

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

- $\mathcal{V}(\texttt{vpkglist})$ *is the smallest set L such that:*

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

- $\mathcal{V}(\texttt{veqpkglist})$ *is the smallest set $L' \subseteq \mathcal{V}(\texttt{vpkglist})$ such that:*

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

Appendix B.2. CUDF formal semantics

CUDF semantics is defined in a style similar to [23], 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 [23] 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.

Appendix B.3. Abstract syntax and semantic domains

The abstract syntax and the semantics is defined using the value domains defined in Appendix B.1. In addition, we give the following definitions:

Definition 2.

- 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 3) and a request (see Definition 5).

Definition 3 (Package description). *A* package description *is a partial function*

$$\begin{aligned}\mathcal{V}(\texttt{ident}) \times \mathcal{V}(\texttt{posint}) &\rightsquigarrow \\ &\mathcal{V}(\texttt{bool}) \times \textsc{Keepvalues} \times \mathcal{V}(\texttt{vpkgformula}) \times \mathcal{V}(\texttt{vpkglist}) \times \mathcal{V}(\texttt{veqpkglist})\end{aligned}$$

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 14 to define the semantics of *package conflicts* in case a package conflicts with itself or a feature provided by the same package.

Definition 4 (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*

$$\begin{array}{rcll} Dom(\phi - (p,n)) & = & Dom(\phi) - \{(p,n)\} & \\ (\phi - (p,n))(q,m) & = & \phi(q,m) & \textit{for all } (q,m) \in Dom(\phi - (p,n)) \end{array}$$

Definition 5 (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.

Appendix B.4. Installations

Definition 6 (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 7 (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

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

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 8 (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

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

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 9 (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*

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

Appendix B.5. Consistent package descriptions

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

Definition 10 (Constraint satisfaction)**.** *The* satisfaction *relation between a natural number n and a constraint $c \in$* Constraints, *noted $n \models c$, is defined as follows:*

$n \models \top$	*for any n*	$n \models (<, v)$	*iff $n < v$*
$n \models (=, v)$	*iff $n = v$*	$n \models (>, v)$	*iff $n > v$*
$n \models (\neq, v)$	*iff $n \neq v$*	$n \models (\leq, v)$	*iff $n \leq v$*
		$n \models (\geq, v)$	*iff $n \geq v$*

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

Definition 11 (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 $p \in \mathcal{V}($`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 12. *Let I be an installation, and $l \in \mathcal{V}($`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 $\mathcal{V}($`veqpkglist`$) \subseteq \mathcal{V}($`vpkglist`$)$, this also defines the satisfaction relation for elements of $\mathcal{V}($`veqpkglist`$)$. Also note that one could transform any $l \in \mathcal{V}($`vpkglist`$)$ into a formula $l_\wedge \in \mathcal{V}($`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 13 (Disjointness)**.** *The* disjointness *relation between an installation I and a set $l \in \mathcal{V}($`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 14. *A package description ϕ is* consistent *if for every package $p \in \mathcal{V}($`ident`$)$ and $n \in i_\phi(p)$ we have that*

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

In the above definition, the first clause corresponds to the *Abundance* property of [23]: all the dependency relations of all installed packages must be satisfied. The second clause corresponds to the *Peace* property of [23]. 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 Appendix B.7).

Appendix B.6. 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.[9]

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

Definition 15 (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 16 (Request semantics)**.** *Let $r = (l_i, l_u, l_d)$ be a request. The* semantics *of r is a relation $\overset{r}{\curvearrowright} \subseteq$ Descr $\times$ Descr defined by $\phi_1 \overset{r}{\curvearrowright} \phi_2$ 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 $n' \in (i_{\phi_1} \cup f_{\phi_1})(p)$.*

[9]The definition of optimization criteria will is outside the scope of this document; see Section 5 of the “Dependency Solving: a Separate Concern in Software Evolution Management” manuscript.

Appendix B.7. Comments on the semantics

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 16. 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 8) 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.[10]

[10]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.