# *aspcud*: A Linux Package Configuration Tool Based on Answer Set Programming

Martin Gebser Roland Kaminski Torsten Schaub*
Universität Potsdam, Institut für Informatik

We present the Linux package configuration tool *aspcud* based on Answer Set Programming. In particular, we detail *aspcud*'s preprocessor turning a CUDF specification into a set of logical facts.

## 1 Introduction

Answer Set Programming (ASP; [3]) owes its increasing popularity as a tool for Knowledge Representation and Reasoning (KRR; [11]) to its attractive combination of a rich yet simple modeling language with high-performance solving capacities. The basic idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an ASP solver for finding answer sets of the program. This approach is closely related to the one pursued in propositional Satisfiability Testing (SAT; [4]), where a given problem is encoded as a propositional theory such that models represent solutions to the problem. Even though, syntactically, ASP programs resemble Prolog programs, they are treated by rather different computational mechanisms, based on advanced Boolean Constraint Satisfaction technology. Albeit SAT and ASP both focus on the generation of propositional models, they differ regarding the semantics of negation, which is classical in SAT and by default in ASP. The built-in completion of "negative knowledge" admits compact problem specifications in ASP, using rules to describe the formation of solution candidates and integrity constraints to deny unintended ones.

Pioneering work on Linux package configuration was done by Tommi Syrjänen in [16], using ASP for representing and solving configuration problems for the Debian GNU/Linux system. Following this tradition, we developed the ASP-based Linux package configuration tool *aspcud*, leveraging modern ASP technology for solving package configuration problems posed in the context of the mancoosi project [13]. As shown in Figure 1, *aspcud* comprises four components, all of which are freely available at [2] (and via [15]). A given specification (in CUDF; [17]) is first preprocessed and mapped to a set of (logical) facts; this step is explained in Section 2. As detailed in Section 3, the facts are then combined with one or more (first-order) ASP encodings of the package configuration problem and jointly passed to the ASP grounder *gringo* [7]. (Our ASP encodings, which are also presented in a companion paper [6] detailing multi-criteria optimization capacities of the ASP solver *clasp* [8] and evaluating them on package configuration problems, are provided here for completeness.) The instantiation of first-order variables upon grounding results in a propositional logic program whose answer sets, representing problem solutions, are in turn computed by *clasp*. The impact of preprocessing on residual problem size as well as solving efficiency is empirically assessed in Section 4. (We do not vary solving strategies here; an experimental comparison between different solving strategies can be found in [5, 6].) Finally, in Section 5, we discuss and compare our methodology with related package configuration approaches.

---

[*] Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Brisbane, Australia.

C. Drescher, I. Lynce and R. Treinen (eds.):
Workshop on Logics for Component Configuration (LoCoCo 2011)
EPTCS 65, 2011, pp. 12–25, doi:10.4204/EPTCS.65.2

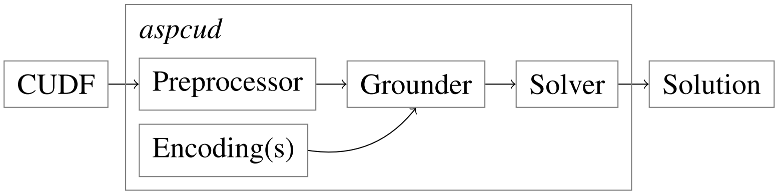

**Figure 1:** Workflow of *aspcud*.

## 2 Preprocessing

Our package configuration tool *aspcud* accepts input in Common Upgradability Description Format (CUDF), developed in the mancoosi project to specify interdependencies of packages belonging to large software distributions. The task of a package manager is to find admissible installations satisfying particular user requests, typically also taking into account soft criteria, such as minimal change of an existing installation. While CUDF admits arithmetic expressions, package formulae, and virtual packages (see below), *aspcud*'s preprocessor generates a flat representation of package interdependencies, so that they can be conveniently handled by the ASP components of *aspcud* taking over afterwards. Below, we give a quick overview of CUDF and optimization criteria, and then describe the generation of ASP facts.

### 2.1 Common Upgradability Description Format (CUDF)

The general schema of a "CUDF document" (with an optional *preamble*; cf. [17]) is as follows:

$$\begin{array}{lllll}
\mathit{preamble} & & & & \\
\textbf{package:}\ \mathtt{name}_1 & \textbf{package:}\ \mathtt{name}_2 & \ldots & \textbf{package:}\ \mathtt{name}_n & \\
\textbf{version:}\ \mathtt{vers}_1 & \textbf{version:}\ \mathtt{vers}_2 & \ldots & \textbf{version:}\ \mathtt{vers}_n & \textbf{request:} \\
\mathit{description}_1 & \mathit{description}_2 & \ldots & \mathit{description}_n & \mathit{description}
\end{array}$$

The pairs $(\mathtt{name}_l, \mathtt{vers}_l)$ for $1 \leq l \leq n$ identify installable packages along with positive integer versions; they must be mutually distinct, that is, $\mathtt{name}_l \neq \mathtt{name}_m$ or $\mathtt{vers}_l \neq \mathtt{vers}_m$ must hold for all $1 \leq l < m \leq n$. Then, the *universe* described by a CUDF document is the set $\mathscr{U} = \{(\mathtt{name}_1, \mathtt{vers}_1), (\mathtt{name}_2, \mathtt{vers}_2), \ldots, (\mathtt{name}_n, \mathtt{vers}_n)\}$ of pairs identifying installable versioned packages.

Each pair $(\mathtt{name}_l, \mathtt{vers}_l)$ can be accompanied with (optional) properties provided in $\mathit{description}_l$. In the most general form, a statement in $\mathit{description}_l$ looks as follows:

$$\mathit{property}\texttt{:}\ p_{1_1} \,|\, p_{2_1} \,|\, \ldots \,|\, p_{k_1}\texttt{,}\ \ p_{1_2} \,|\, p_{2_2} \,|\, \ldots \,|\, p_{k_2}\texttt{,}\ \ \ldots\texttt{,}\ \ p_{1_m} \,|\, p_{2_m} \,|\, \ldots \,|\, p_{k_m}$$

In such a statement, $\mathit{property} \in \{\texttt{conflicts}, \texttt{depends}, \texttt{recommends}, \texttt{provides}\}$ determines a kind of package interdependency, '`|`' and '`,`' stand for disjunction and conjunction, respectively, and $p_{j_i}$ for $1 \leq i \leq m, 1 \leq j_i \leq k_i$ is an expression of the form '`name [`$\mathit{op}$ $n$`]`', in which $\mathit{op} \in \{=, !=, <, <=, >, >=\}$ denotes an (optional) arithmetic operation along with a positive integer $n$. Moreover, if '`installed: true`' is provided in $\mathit{description}_l$ for $1 \leq l \leq n$, it means that package $\mathtt{name}_l$ in version $\mathtt{vers}_l$ belongs to an *existing installation*, and we denote the set of all such pairs $(\mathtt{name}_l, \mathtt{vers}_l)$ by $\mathscr{O}$.

For a $\mathit{property} \in \{\texttt{install}, \texttt{remove}, \texttt{upgrade}\}$ in the *description* below the keyword '**`request:`**', for uniformity, we assume the same syntax as with package *property* statements considered before.[1] The requested properties describe goals that must be satisfied by a *follow-up installa-*

---

[1] The specification of CUDF [17] is more restrictive by not allowing for disjunction in package formulae associated with $\mathit{property} \in \{\texttt{conflicts}, \texttt{provides}, \texttt{install}, \texttt{remove}, \texttt{upgrade}\}$. Moreover, note that CUDF additionally admits `keep` as

```
package:     inst        package:     feat        package:     recomm
version:     3           version:     1           version:     1
conflicts:   conf < 3    provides:    conf = 3    conflicts:   option
package:     inst
version:     2                                    package:     option
depends:     dep < 2     package:     dep         version:     1
package:     inst        version:     3           depends:     avail
version:     1           conflicts:   dep
depends:     dep         recommends:  recomm      package:     avail
                         package:     dep         version:     1
package:     conf        version:     2           installed:   true
version:     2           conflicts:   dep < 2
package:     conf        package:     dep         request:
version:     1           version:     1           install:     inst
installed:   true        installed:   true        upgrade:     conf > 1
```

**Figure 2:** CUDF document specifying the (non-empty) interdependencies $\mathit{Targets}(\mathtt{inst},3,\mathtt{conflicts}) = [\{(\mathtt{conf},1),(\mathtt{conf},2)\}]$, $\mathit{Targets}(\mathtt{inst},2,\mathtt{depends}) = [\{(\mathtt{dep},1)\}]$, $\mathit{Targets}(\mathtt{inst},1,\mathtt{depends}) = [\{(\mathtt{dep},n) \mid n \in \mathbb{N}\}]$, $\mathit{Targets}(\mathtt{feat},1,\mathtt{provides}) = [\{(\mathtt{conf},3)\}]$, $\mathit{Targets}(\mathtt{dep},3,\mathtt{conflicts}) = [\{(\mathtt{dep},n) \mid n \in \mathbb{N}\}]$, $\mathit{Targets}(\mathtt{dep},3,\mathtt{recommends}) = [\{(\mathtt{recomm},n) \mid n \in \mathbb{N}\}]$, $\mathit{Targets}(\mathtt{dep},2,\mathtt{conflicts}) = [\{(\mathtt{dep},1)\}]$, $\mathit{Targets}(\mathtt{recomm},1,\mathtt{conflicts}) = [\{(\mathtt{option},n) \mid n \in \mathbb{N}\}]$, and $\mathit{Targets}(\mathtt{option},1,\mathtt{depends}) = [\{(\mathtt{avail},n) \mid n \in \mathbb{N}\}]$; (non-empty) request targets consist of $\mathit{Targets}(\mathtt{install}) = [\{(\mathtt{inst},n) \mid n \in \mathbb{N}\}]$ and $\mathit{Targets}(\mathtt{upgrade}) = [\{(\mathtt{conf},n) \mid n \in \mathbb{N}, n > 1\}]$.

*tion* $\mathcal{P}$, where certain versioned packages might have to be installed, removed, or upgraded, respectively.

In order to abstract from arithmetic expressions admitted in CUDF, for '$\mathtt{name}\ [\mathit{op}\ n]$', we define:

$$\mathit{targets}(\mathtt{name}\ [\mathit{op}\ n]) = \begin{cases} \{(\mathtt{name},n) \mid n \in \mathbb{N}^+ \text{ such that } (n\ \mathit{op}\ n) \text{ holds}\} & \text{if } \mathit{op}\ n \text{ is specified} \\ \{(\mathtt{name},n) \mid n \in \mathbb{N}^+\} & \text{if } \mathit{op}\ n \text{ is omitted} \end{cases}$$

We extend the notion of targets to package formulae associated with some $\mathit{property} \in \{\mathtt{conflicts}, \mathtt{depends}, \mathtt{recommends}, \mathtt{provides}, \mathtt{install}, \mathtt{remove}, \mathtt{upgrade}\}$ by defining the following multiset:[2]

$$\mathit{Targets}(\mathit{property}) = [\mathit{targets}(p_{1_i}) \cup \mathit{targets}(p_{2_i}) \cup \cdots \cup \mathit{targets}(p_{k_i}) \mid 1 \leq i \leq m]$$

Moreover, let $\mathit{Targets}(\mathtt{name}_l, \mathtt{vers}_l, \mathit{property})$ be $\mathit{Targets}(\mathit{property})$ for $(\mathtt{name}_l, \mathtt{vers}_l) \in \mathcal{U}$ and $\mathit{property} \in \{\mathtt{conflicts}, \mathtt{depends}, \mathtt{recommends}, \mathtt{provides}\}$, where either a unique package formula is provided for $\mathit{property}$ in $\mathit{description}_l$, or $\mathit{Targets}(\mathit{property}) = \emptyset$ if $\mathit{property}$ is not specified in $\mathit{description}_l$. Likewise, we let $\mathit{Targets}(\mathit{property}) = \emptyset$ for $\mathit{property} \in \{\mathtt{install}, \mathtt{remove}, \mathtt{upgrade}\}$ if no corresponding statement is provided in the $\mathit{description}$ below '**`request:`**', while the package formula defining $\mathit{property}$ must be unique otherwise.

As an example, consider the CUDF document shown in Figure 2. The existing installation, marked via '`installed: true`', is $\mathcal{O} = \{(\mathtt{conf},1), (\mathtt{dep},1), (\mathtt{avail},1)\}$. The universe, including all versioned packages, is $\mathcal{U} = \mathcal{O} \cup \{(\mathtt{inst},3), (\mathtt{inst},2), (\mathtt{inst},1), (\mathtt{conf},2), (\mathtt{feat},1), (\mathtt{dep},3), (\mathtt{dep},2), (\mathtt{recomm},1), (\mathtt{option},1)\}$. The CUDF document further specifies the (non-empty) multisets of targets of package interdependencies and requests, respectively, provided in the caption of Figure 2; their particular meanings are described below in the context of ASP fact generation.

---

$\mathit{property}$ in $\mathit{description}_l$ for $1 \leq l \leq n$, which we omitted here because it is straightforward to map `keep` to `install`.

[2] Multisets are needed to reflect optimization criteria dealing with (un)satisfied recommendations, below collected in $\mathbf{R}^{\mathcal{P}}_{\mathcal{U}}$.

## 2.2 Optimization Criteria

The preprocessor of *aspcud* takes *optimization criteria* evaluated in competitions by mancoosi [13] into account. Given a universe $\mathscr{U}$, an existing installation $\mathscr{O}$, and a follow-up installation $\mathscr{P}$, such criteria rely on the minimization or maximization of the following sets:

$$\begin{aligned}
\mathbf{N}_{\mathscr{O}}^{\mathscr{P}} &= \{\texttt{name} \mid (\texttt{name},\texttt{vers}) \in \mathscr{P}, \{(\texttt{name},n) \mid n \in \mathbb{N}\} \cap \mathscr{O} = \emptyset\} \\
\mathbf{D}_{\mathscr{O}}^{\mathscr{P}} &= \{\texttt{name} \mid (\texttt{name},\texttt{vers}) \in \mathscr{O}, \{(\texttt{name},n) \mid n \in \mathbb{N}\} \cap \mathscr{P} = \emptyset\} \\
\mathbf{C}_{\mathscr{O}}^{\mathscr{P}} &= \{\texttt{name} \mid (\texttt{name},\texttt{vers}) \in (\mathscr{P} \setminus \mathscr{O}) \cup (\mathscr{O} \setminus \mathscr{P})\} \\
\mathbf{U}_{\mathscr{U}}^{\mathscr{P}} &= \{\texttt{name} \mid (\texttt{name},\texttt{vers}) \in \mathscr{P}, (\texttt{name}, \max\{n \mid (\texttt{name},n) \in \mathscr{U}\}) \notin \mathscr{P}\} \\
\mathbf{R}_{\mathscr{U}}^{\mathscr{P}} &= \{(\texttt{name},\texttt{vers},i) \mid (\texttt{name},\texttt{vers}) \in \mathscr{P}, R_i \cap \mathit{Provide}(\mathscr{P}) = \emptyset, \\
&\qquad\qquad \mathit{Targets}(\texttt{name},\texttt{vers},\texttt{recommends}) = [R_1,\dots,R_i,\dots,R_m]\}
\end{aligned}$$

Here, $\mathbf{N}_{\mathscr{O}}^{\mathscr{P}}$ is the collection of packages `name` such that some version `vers` belongs to $\mathscr{P}$, while $\mathscr{O}$ contains no pair $(\texttt{name},n)$; that is, package `name` is new in the follow-up installation $\mathscr{P}$. Similarly, $\mathbf{D}_{\mathscr{O}}^{\mathscr{P}}$ and $\mathbf{C}_{\mathscr{O}}^{\mathscr{P}}$ collect packages `name` that are deleted or changed, respectively, where change means that some version `vers` of `name` is new or deleted in the transition from $\mathscr{O}$ to $\mathscr{P}$. The sets $\mathbf{U}_{\mathscr{U}}^{\mathscr{P}}$ and $\mathbf{R}_{\mathscr{U}}^{\mathscr{P}}$ investigate the follow-up installation $\mathscr{P}$ relative to the universe $\mathscr{U}$. A package `name` belongs to $\mathbf{U}_{\mathscr{U}}^{\mathscr{P}}$ if, for each pair $(\texttt{name},\texttt{vers})$ in $\mathscr{P}$, there is some $(\texttt{name},n)$ in $\mathscr{U}$ such that $\texttt{vers} < n$; that is, the latest version of `name` is missing in $\mathscr{P}$. Finally, a triple $(\texttt{name},\texttt{vers},i)$ in $\mathbf{R}_{\mathscr{U}}^{\mathscr{P}}$ points to a disjunction '$p_{1_i} \mid p_{2_i} \mid \dots \mid p_{k_i}$' in the `recommends` statement associated with $(\texttt{name},\texttt{vers})$ such that $\mathscr{P}$ neither contains nor provides any element of $\mathit{targets}(p_{1_i}) \cup \mathit{targets}(p_{2_i}) \cup \dots \cup \mathit{targets}(p_{k_i})$. In fact, by $\mathit{Provide}(\mathscr{P}) = \bigcup_{(\texttt{name},\texttt{vers}) \in \mathscr{P}} \mathit{Provide}(\texttt{name},\texttt{vers})$ and $\mathit{Provide}(\texttt{name},\texttt{vers}) = \{(\texttt{name},\texttt{vers})\} \cup (\bigcup_{P \in \mathit{Targets}(\texttt{name},\texttt{vers},\texttt{provides})} P)$, we refer to the union of $\mathscr{P}$ and the targets of its packages' `provides` statements. This allows us to abstract from "virtual packages" that may not be installable themselves, but can be provided by other packages. Note that installable and virtual packages are not necessarily disjoint; e.g., the CUDF document in Figure 2 specifies version `1` and `2` of `conf` as installable, while version `3` is provided by `(feat,1)`. In the following, we indicate the objective of *maximizing* or *minimizing* the *cardinality* of any of the sets $\mathbf{O}_{\mathscr{O}/\mathscr{U}}^{\mathscr{P}}$ defined above by writing $+\mathbf{O}_{\mathscr{O}/\mathscr{U}}^{\mathscr{P}}$ or $-\mathbf{O}_{\mathscr{O}/\mathscr{U}}^{\mathscr{P}}$, respectively.

## 2.3 Generation of ASP Facts

We are now ready to specify the algorithm applied by *aspcud*'s preprocessor to compute the transitive closure $\mathscr{C}$ of versioned packages that may belong to a follow-up installation $\mathscr{P}$. The general idea is to include versioned packages by need, that is, if they are among the targets of some `install` or `upgrade` request, a `depends` statement, or may otherwise serve some user-specified objective. (E.g., $+\mathbf{N}_{\mathscr{O}}^{\mathscr{P}}$ describes the objective of installing as many new packages as possible, so that all pairs $(\texttt{name},\texttt{vers})$ in $\mathscr{U}$ such that `name` does not occur in $\mathscr{O}$ would be added to $\mathscr{C}$.) Given a universe $\mathscr{U}$, an existing installation $\mathscr{O}$, and a set $\mathbf{O} \subseteq \{+\mathbf{N}_{\mathscr{O}}^{\mathscr{P}}, -\mathbf{N}_{\mathscr{O}}^{\mathscr{P}}, +\mathbf{D}_{\mathscr{O}}^{\mathscr{P}}, -\mathbf{D}_{\mathscr{O}}^{\mathscr{P}}, +\mathbf{C}_{\mathscr{O}}^{\mathscr{P}}, -\mathbf{C}_{\mathscr{O}}^{\mathscr{P}}, +\mathbf{U}_{\mathscr{U}}^{\mathscr{P}}, -\mathbf{U}_{\mathscr{U}}^{\mathscr{P}}, +\mathbf{R}_{\mathscr{U}}^{\mathscr{P}}, -\mathbf{R}_{\mathscr{U}}^{\mathscr{P}}\}$ of objectives, the transitive closure $\mathscr{C}$ is computed via Algorithm 1.

In Line 1 of Algorithm 1, "negative" requests given by `remove` and also `upgrade` are evaluated; packages that must not be installed are collected in *Out* to exclude their addition to $\mathscr{C}$ in the sequel. While exclusions due to `remove` statements are straightforward (any package fulfilling some `remove` target must not be installed), the issue becomes more involved with `upgrade`. On the one hand, any element of $\mathit{Targets}(\texttt{upgrade})$ resembles an `install` request because it must be served by some package (directly or via a provided virtual package) in a follow-up installation $\mathscr{P}$. On the other hand, there are

```
1  $\mathit{Out} \leftarrow \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \mid D \in \mathit{Targets}(\mathtt{remove}), D \cap \mathit{Provide}(\mathtt{name},\mathtt{vers}) \neq \emptyset\}$
       $\cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \mid U \in \mathit{Targets}(\mathtt{upgrade}), (\mathtt{name}',m) \in U,$
           $(\mathtt{name}',n) \in \mathit{Provide}(\mathtt{name},\mathtt{vers}), (\mathtt{name}',n') \in \mathit{Provide}(\mathscr{O}), n < n'\}$
       $\cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \mid U \in \mathit{Targets}(\mathtt{upgrade}),$
           $1 < |\{(\mathtt{name}',n) \in \mathit{Provide}(\mathtt{name},\mathtt{vers}) \mid (\mathtt{name}',m) \in U\}|\}$
       $\cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \mid U \in \mathit{Targets}(\mathtt{upgrade}), U \cap \mathit{Provide}(\mathtt{name},\mathtt{vers}) = \emptyset,$
           $\{\mathtt{name}' \mid (\mathtt{name}',m) \in U\} \cap \{\mathtt{name}' \mid (\mathtt{name}',n) \in \mathit{Provide}(\mathtt{name},\mathtt{vers})\} \neq \emptyset\}$
2  if $\{I \in \mathit{Targets}(\mathtt{install}) \cup \mathit{Targets}(\mathtt{upgrade}) \mid I \cap \mathit{Provide}(\mathscr{U} \setminus \mathit{Out}) = \emptyset\} \neq \emptyset$ then return $\emptyset$
3  $\mathscr{C} \leftarrow \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid I \in \mathit{Targets}(\mathtt{install}) \cup \mathit{Targets}(\mathtt{upgrade}), I \cap \mathit{Provide}(\mathtt{name},\mathtt{vers}) \neq \emptyset\}$
4  if $+\mathbf{N}^{\mathscr{P}}_{\mathscr{O}} \in \mathbf{O}$ then $\mathscr{C} \leftarrow \mathscr{C} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid \{n \mid (\mathtt{name},n) \in \mathscr{O}\} = \emptyset\}$
5  if $-\mathbf{D}^{\mathscr{P}}_{\mathscr{O}} \in \mathbf{O}$ then $\mathscr{C} \leftarrow \mathscr{C} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid \{n \mid (\mathtt{name},n) \in \mathscr{O}\} \neq \emptyset\}$
6  if $+\mathbf{C}^{\mathscr{P}}_{\mathscr{O}} \in \mathbf{O}$ then $\mathscr{C} \leftarrow \mathscr{C} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid (\mathtt{name},\mathtt{vers}) \notin \mathscr{O}\}$
7  if $-\mathbf{C}^{\mathscr{P}}_{\mathscr{O}} \in \mathbf{O}$ then $\mathscr{C} \leftarrow \mathscr{C} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid (\mathtt{name},\mathtt{vers}) \in \mathscr{O}\}$
8  if $+\mathbf{U}^{\mathscr{P}}_{\mathscr{U}} \in \mathbf{O}$ then $\mathscr{C} \leftarrow \mathscr{C} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid \mathtt{vers} < \max\{n \mid (\mathtt{name},n) \in \mathscr{U}\}\}$
9  if $+\mathbf{R}^{\mathscr{P}}_{\mathscr{U}} \in \mathbf{O}$ then $\mathscr{C} \leftarrow \mathscr{C} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus \mathit{Out} \mid \mathit{Targets}(\mathtt{name},\mathtt{vers},\mathtt{recommends}) \neq \emptyset\}$
10 repeat
11     $\mathit{Add} \leftarrow \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus (\mathit{Out} \cup \mathscr{C}) \mid (\mathtt{name}',\mathtt{vers}') \in \mathscr{C},$
           $D \in \mathit{Targets}(\mathtt{name}',\mathtt{vers}',\mathtt{depends}), D \cap \mathit{Provide}(\mathtt{name},\mathtt{vers}) \neq \emptyset\}$
12     if $-\mathbf{R}^{\mathscr{P}}_{\mathscr{U}} \in \mathbf{O}$ then $\mathit{Add} \leftarrow \mathit{Add} \cup \{(\mathtt{name},\mathtt{vers}) \in \mathscr{U} \setminus (\mathit{Out} \cup \mathscr{C}) \mid (\mathtt{name}',\mathtt{vers}') \in \mathscr{C},$
           $R \in \mathit{Targets}(\mathtt{name}',\mathtt{vers}',\mathtt{recommends}), R \cap \mathit{Provide}(\mathtt{name},\mathtt{vers}) \neq \emptyset\}$
13     if $-\mathbf{U}^{\mathscr{P}}_{\mathscr{U}} \in \mathbf{O}$ then $\mathit{Add} \leftarrow \mathit{Add} \cup \{(\mathtt{name},\max\{n \mid (\mathtt{name},n) \in \mathscr{U}\}) \in \mathscr{U} \setminus (\mathit{Out} \cup \mathscr{C}) \mid$
           $(\mathtt{name},\mathtt{vers}) \in \mathscr{C}\}$
14     $\mathscr{C} \leftarrow \mathscr{C} \cup \mathit{Add}$
15 until $\mathit{Add} = \emptyset$
16 return $\mathscr{C}$
```

**Algorithm 1:** Compute transitive closure $\mathscr{C}$ wrt. universe $\mathscr{U}$, existing installation $\mathscr{O}$, and objectives $\mathbf{O}$.

three additional requirements, which can make the installation of particular packages prohibitive. First, the version number of packages subject to `upgrade` must in a follow-up installation $\mathscr{P}$ not be smaller than in the existing installation $\mathscr{O}$ (if some version is provided by $\mathscr{O}$). Second, exactly one version must be available in $\mathscr{P}$, so that packages providing several versions at once cannot belong to $\mathscr{P}$. Third, the `install` request implied by an `upgrade` target along with the unique version requirement prohibit the installation of packages providing only non-matching versions. These three conditions are taken into account to reflect `upgrade` requests in *Out*.[3] (For the CUDF document in Figure 2, `(conf,2)` and `(feat,1)` can fulfill the target of the `upgrade` request '`conf > 1`', while `(conf,1)` is excluded in view of its non-matching version.) Given the set *Out* of packages that must not belong to a follow-up installation $\mathscr{P}$, the test in Line 2 of Algorithm 1 identifies cases in which `install` or `upgrade` targets remain unsatisfiable, regardless of further preprocessing, so that $\emptyset$ can be immediately returned.

Provided that the test in Line 2 failed, packages not in *Out* that may serve some `install` or `upgrade` target are used to initialize the transitive closure $\mathscr{C}$ in Line 3. In Line 4–9, $\mathscr{C}$ is further extended in view of the objectives in $\mathbf{O}$. As already mentioned, it might be desirable to install any version of a package `name` not occurring in the existing installation $\mathscr{O}$ if $+\mathbf{N}^{\mathscr{P}}_{\mathscr{O}}$ belongs to $\mathbf{O}$, describing the objective of installing as many new packages as possible; if so, $\mathscr{C}$ is extended accordingly in Line 4. Note that the objectives

---

[3] The CUDF specification [17] disallows disjunction in `upgrade` requests, and we here generalize `upgrade` targets to disjunction in an "arbitrary" way. However, in the case without disjunction, the packages included in *Out* due to an `upgrade` target cannot belong to a follow-up installation $\mathscr{P}$ according to the semantics given in [17].

of the form $+\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}$ are useless in practice, as they favor follow-up installations $\mathscr{P}$ that are as different from $\mathscr{O}$, or as suboptimal regarding latest versions or `recommends` targets as possible. However, such "anti-optimization" would in principle be allowed in the *user* track of competitions by mancoosi, and thus Algorithm 1 includes cases to extend $\mathscr{C}$ accordingly. The reasonable cases in Line 5 and 7 apply if package removals or changes, respectively, are to be minimized, so that it may help to add all (installed) versions of packages `name` occurring in $\mathscr{O}$ to $\mathscr{C}$. For instance, if $-\mathbf{D}^{\mathscr{P}}_{\mathscr{O}}$, aiming at the minimization of package removals, belongs to $\mathbf{O}$, `(conf,2)`, `(dep,3)`, `(dep,2)`, `(dep,1)`, and `(avail,1)` are added to $\mathscr{C}$ in Line 5 for the CUDF document in Figure 2, given that `(conf,1)`, `(dep,1)`, and `(avail,1)` are installed in $\mathscr{O}$. Note that the installed pair `(conf,1)` is not added to $\mathscr{C}$, as `(conf,1)` belongs to *Out*.

After its initialization wrt. requests (Line 3) and objectives (Line 4–9), the transitive closure $\mathscr{C}$ is successively extended in the loop in Line 10–15 of Algorithm 1. To this end, packages `(name,vers)` matching some dependency of elements already in $\mathscr{C}$ are collected in Line 11, provided that the installation of `(name,vers)` is not excluded by *Out*. Similarly, packages serving `recommends` statements of elements in $\mathscr{C}$ are collected in Line 12, but only if the minimization of unsatisfied recommendations is requested via the objective $-\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}$. Finally, if packages ought to be installed in their latest versions, as it can be specified via $-\mathbf{U}^{\mathscr{P}}_{\mathscr{U}}$, we also collect such latest versions in Line 13. The three cases justifying the addition of packages to $\mathscr{C}$ are applied until saturation, and the obtained fixpoint is returned in Line 16. Any package remaining in $\mathscr{U}\setminus\mathscr{C}$ belongs to *Out*, meaning that it must not be installed, or is irrelevant regarding dependencies, requests, and objectives. Hence, packages outside $\mathscr{C}$ need not be reflected in ASP facts (described below), so that both instance and residual problem size can be reduced. For the CUDF document in Figure 2, assuming that the objective $-\mathbf{D}^{\mathscr{P}}_{\mathscr{O}}$ is provided in $\mathbf{O}$, $\mathscr{C}$ is initialized with

- `(inst,3)`, `(inst,2)`, and `(inst,1)` in view of the request '`install: inst`',
- `(conf,2)` and `(feat,1)` in order to serve '`upgrade: conf` > `1`', and additionally
- `(dep,3)`, `(dep,2)`, `(dep,1)`, and `(avail,1)` due to the objective $-\mathbf{D}^{\mathscr{P}}_{\mathscr{O}}$.

While tracking the dependencies of these packages does not contribute any further elements to $\mathscr{C}$, if the objective $-\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}$ is given in $\mathbf{O}$, '`recommends: recomm`' associated with `(dep,3)` justifies the addition of `(recomm,1)` to $\mathscr{C}$. The packages still outside $\mathscr{C}$ are `(conf,1)`, which is excluded due to the provided `upgrade` request, and `(option,1)`, as it does not support any element of $\mathscr{C}$ and could thus be included only if some of the objectives $+\mathbf{N}^{\mathscr{P}}_{\mathscr{O}}$ and $+\mathbf{C}^{\mathscr{P}}_{\mathscr{O}}$ would reward new packages or changes, respectively.

Given the transitive closure $\mathscr{C}$ of relevant packages, the final step of *aspcud*'s preprocessor is to generate a representation of package interdependencies, requests, and objectives in terms of ASP facts. Note that, in competitions by mancoosi, objectives are lexicographically ordered by significance; hence, we below identify $\mathbf{O}$ with a sequence of objectives, written as $(\#_1[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_1,\dots,\#_n[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_n)$ in increasing order of significance, where $\#_i \in \{+,-\}$ and $[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i \in \{\mathbf{N}^{\mathscr{P}}_{\mathscr{O}},\mathbf{D}^{\mathscr{P}}_{\mathscr{O}},\mathbf{C}^{\mathscr{P}}_{\mathscr{O}},\mathbf{U}^{\mathscr{P}}_{\mathscr{U}},\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}\}$ for $1 \leq i \leq n$. We further associate some ASP constant $c_{\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}}$ with each $\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}$ (`newpackage` for $\mathbf{O}^{\mathscr{P}}_{\mathscr{O}} = \mathbf{N}^{\mathscr{P}}_{\mathscr{O}}$, `remove` for $\mathbf{O}^{\mathscr{P}}_{\mathscr{O}} = \mathbf{D}^{\mathscr{P}}_{\mathscr{O}}$, `change` for $\mathbf{O}^{\mathscr{P}}_{\mathscr{O}} = \mathbf{C}^{\mathscr{P}}_{\mathscr{O}}$, `uptodate` for $\mathbf{O}^{\mathscr{P}}_{\mathscr{U}} = \mathbf{U}^{\mathscr{P}}_{\mathscr{U}}$, and `recommend` for $\mathbf{O}^{\mathscr{P}}_{\mathscr{U}} = \mathbf{R}^{\mathscr{P}}_{\mathscr{U}}$). Moreover, for any set $P$ of packages, we write $id_P$ to refer to some ASP constant associated with the set $P$, where $id_P \neq id_{P'}$ if $P \neq P'$. Then, the facts obtained for a CUDF document (specifying a universe $\mathscr{U}$ and an existing installation $\mathscr{O}$), a sequence $\mathbf{O}$ of objectives, and $\mathscr{C}$ are collected in $\pi$ as shown in Figure 3.

In Figure 3, the subset $\tau$ of $\pi$ groups packages fulfilling targets of package interdependencies or requests in sets $P$, and respective facts introduce constants $id_P$ referring to $P$. While facts over the predicate `depends` in (1) simply link the targets of dependencies to packages that provide them, `recommends` in (2) introduces a counter $r$ along with each set $P$ of packages fulfilling a recommendation $R_i$ because several elements of the multiset *Targets*(`name`,`vers`,`recommends`) $= [R_1,\dots,R_i,\dots,R_m]$ may share the

$$\begin{aligned}
\tau \;=\; & \{\mathbf{\texttt{depends}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{C}, D \in \mathit{Targets}(\texttt{name},\texttt{vers},\mathbf{\texttt{depends}}),\\
& P = \{(\texttt{name}',\texttt{vers}') \in \mathscr{C} \mid D \cap \mathit{Provide}(\texttt{name}',\texttt{vers}') \neq \emptyset\}\}
\end{aligned} \tag{1}$$

$$\begin{aligned}
\cup \;\; & \{\mathbf{\texttt{recommends}}\texttt{(name,vers,}\mathit{id}_P,r\texttt{).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{C}, \{+\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}, -\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}\} \cap \mathbf{O} \neq \emptyset,\\
& \mathit{Targets}(\texttt{name},\texttt{vers},\mathbf{\texttt{recommends}}) = [R_1,\ldots,R_i,\ldots,R_m],\\
& P = \{(\texttt{name}',\texttt{vers}') \in \mathscr{C} \mid R_i \cap \mathit{Provide}(\texttt{name}',\texttt{vers}') \neq \emptyset\},\\
& r = |\{1 \leq j \leq m \mid \{(\texttt{name}',\texttt{vers}') \in \mathscr{C} \mid R_j \cap \mathit{Provide}(\texttt{name}',\texttt{vers}') \neq \emptyset\} = P\}|\}
\end{aligned} \tag{2}$$

$$\begin{aligned}
\cup \;\; & \{\mathbf{\texttt{conflict}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{C}, C = \textstyle\bigcup_{T \in \mathit{Targets}(\texttt{name},\texttt{vers},\mathbf{\texttt{conflicts}})} T,\\
& \emptyset \subset P = \{(\texttt{name}',\texttt{vers}') \in \mathscr{C} \setminus \{(\texttt{name},\texttt{vers})\} \mid C \cap \mathit{Provide}(\texttt{name}',\texttt{vers}') \neq \emptyset\}\}
\end{aligned} \tag{3}$$

$$\begin{aligned}
\cup \;\; & \{\mathbf{\texttt{conflict}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{C},\\
& U \in \mathit{Targets}(\mathbf{\texttt{upgrade}}), U \cap \mathit{Provide}(\texttt{name},\texttt{vers}) \neq \emptyset,\\
& \emptyset \subset P = \{(\texttt{name}',\texttt{vers}') \in \mathscr{C} \mid U \cap \mathit{Provide}(\texttt{name}',\texttt{vers}') \neq \emptyset,\\
& \qquad\qquad U \cap \mathit{Provide}(\texttt{name}',\texttt{vers}') \neq U \cap \mathit{Provide}(\texttt{name},\texttt{vers})\}\}
\end{aligned} \tag{4}$$

$$\begin{aligned}
\cup \;\; & \{\mathbf{\texttt{request}}\texttt{(}\mathit{id}_P\texttt{).} \mid I \in \mathit{Targets}(\mathbf{\texttt{install}}) \cup \mathit{Targets}(\mathbf{\texttt{upgrade}}),\\
& P = \{(\texttt{name},\texttt{vers}) \in \mathscr{C} \mid I \cap \mathit{Provide}(\texttt{name},\texttt{vers}) \neq \emptyset\}\}
\end{aligned} \tag{5}$$

$$\pi \;=\; \tau$$

$$\cup \;\; \{\mathbf{\texttt{satisfies}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in P, (\mathbf{\texttt{depends}}\texttt{(name}',\texttt{vers}',\mathit{id}_P\texttt{).}) \in \tau\} \tag{6}$$

$$\begin{aligned}
\cup \;\; & \{\mathbf{\texttt{satisfies}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in P,\\
& \qquad\qquad (\mathbf{\texttt{recommends}}\texttt{(name}',\texttt{vers}',\mathit{id}_P,r\texttt{).}) \in \tau\}
\end{aligned} \tag{7}$$

$$\cup \;\; \{\mathbf{\texttt{satisfies}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in P, (\mathbf{\texttt{conflict}}\texttt{(name}',\texttt{vers}',\mathit{id}_P\texttt{).}) \in \tau\} \tag{8}$$

$$\cup \;\; \{\mathbf{\texttt{satisfies}}\texttt{(name,vers,}\mathit{id}_P\texttt{).} \mid (\texttt{name},\texttt{vers}) \in P, (\mathbf{\texttt{request}}\texttt{(}\mathit{id}_P\texttt{).}) \in \tau\} \tag{9}$$

$$\cup \;\; \{\mathbf{\texttt{unit}}\texttt{(name,vers).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{C}\} \tag{10}$$

$$\cup \;\; \{\mathbf{\texttt{installed}}\texttt{(name,vers).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{O}\} \tag{11}$$

$$\cup \;\; \{\mathbf{\texttt{newestversion}}\texttt{(name, }\max\{n \mid (\texttt{name},n) \in \mathscr{U}\}\texttt{).} \mid (\texttt{name},\texttt{vers}) \in \mathscr{C}\} \tag{12}$$

$$\cup \;\; \{\mathbf{\texttt{criterion}}\texttt{(}c_{[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i}\texttt{,}\#_i i\texttt{).} \mid \mathbf{O} = (\#_1[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_1,\ldots,\#_n[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_n), 1 \leq i \leq n\} \tag{13}$$

**Figure 3:** ASP facts for a CUDF document, a sequence **O** of objectives, and a set $\mathscr{C} \subseteq \mathscr{U}$ of packages.

same providers $P$. Also note that (2) contributes facts to $\tau$ (and $\pi$) only if $\#\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}$ for $\# \in \{+,-\}$ is among the objectives in **O**. The packages $P$ considered by `conflict` in (3) are obtained by joining all $T \in \mathit{Targets}(\texttt{name},\texttt{vers},\mathbf{\texttt{conflicts}})$ in $C$ before collecting their providers in $P$. Note that $(\texttt{name},\texttt{vers})$ can by definition (cf. [17]) not be in conflict with itself, even if it fulfills some $T \in \mathit{Targets}(\texttt{name},\texttt{vers},\mathbf{\texttt{conflicts}})$; this situation arises with $(\texttt{dep},3)$ in Figure 2, where ‘`conflicts: dep`’ specifies a universal conflict with any version of `dep` (and packages including `dep` in their `provides` statements). Additional conflicts may be induced by `upgrade` requests in view of their unique version requirement, and thus packages providing different elements of some $U \in \mathit{Targets}(\mathbf{\texttt{upgrade}})$ are marked as conflicting via (4); for instance, the `upgrade` request ‘`conf > 1`’ in Figure 2 is reflected by facts ‘$\mathbf{\texttt{conflict}}\texttt{(conf,2,}\mathit{id}_{\{(\texttt{feat},1)\}}\texttt{).}$’ and ‘$\mathbf{\texttt{conflict}}\texttt{(feat,1,}\mathit{id}_{\{(\texttt{conf},2)\}}\texttt{).}$’, obtained because $(\texttt{feat},1)$ provides $(\texttt{conf},3)$ (as a virtual package). Finally, facts over the predicate `request` in (5) group packages $P$ fulfilling `install` or `upgrade` requests to express that some element of $P$ must be included in a follow-up installation $\mathscr{P}$. Note that all packages referred to in facts of $\tau$, via $(\texttt{name},\texttt{vers})$ in arguments or belonging to $P$ associated with some constant $\mathit{id}_P$, are elements of the transitive closure $\mathscr{C}$; that is, the package interdependencies and requests specified by $\tau$ are limited to $\mathscr{C}$.

The full ASP instance $\pi$ extracted from a CUDF document is obtained by joining $\tau$ with further facts. The first group of them, given in (6)–(9) in Figure 3, links packages $(\texttt{name},\texttt{vers}) \in P$ to $\mathit{id}_P$ via the

```
unit(inst,3).
conflict(inst,3,id_{(conf,2)}).
unit(inst,2).
depends(inst,2,id_{(dep,1)}).
unit(inst,1).
depends(inst,1,id_{(dep,3),(dep,2),(dep,1)}).
newestversion(inst,3).

unit(conf,2).
conflict(conf,2,id_{(feat,1)}).
newestversion(conf,2).
installed(conf,1).

unit(feat,1).
conflict(feat,1,id_{(conf,2)}).
newestversion(feat,1).

unit(dep,3).
conflict(dep,3,id_{(dep,2),(dep,1)}).
unit(dep,2).
conflict(dep,2,id_{(dep,1)}).
unit(dep,1).
newestversion(dep,3).
installed(dep,1).

unit(avail,1).
newestversion(avail,1).
installed(avail,1).

request(id_{(inst,3),(inst,2),(inst,1)}).
request(id_{(conf,2),(feat,1)}).

satisfies(conf,2,id_{(conf,2)}).
satisfies(dep,1,id_{(dep,1)}).
satisfies(dep,3,id_{(dep,3),(dep,2),(dep,1)}).
satisfies(dep,2,id_{(dep,3),(dep,2),(dep,1)}).
satisfies(dep,1,id_{(dep,3),(dep,2),(dep,1)}).
satisfies(feat,1,id_{(feat,1)}).
satisfies(dep,2,id_{(dep,2),(dep,1)}).
satisfies(dep,1,id_{(dep,2),(dep,1)}).
satisfies(inst,3,id_{(inst,3),(inst,2),(inst,1)}).
satisfies(inst,2,id_{(inst,3),(inst,2),(inst,1)}).
satisfies(inst,1,id_{(inst,3),(inst,2),(inst,1)}).
satisfies(conf,2,id_{(conf,2),(feat,1)}).
satisfies(feat,1,id_{(conf,2),(feat,1)}).

criterion(change,-1).
criterion(remove,-2).
```

**Figure 4:** ASP facts $\pi$ obtained for the CUDF document in Figure 2 along with $\mathbf{O} = (-\mathbf{C}^{\mathscr{P}}_{\mathscr{O}}, -\mathbf{D}^{\mathscr{P}}_{\mathscr{O}})$.

predicate `satisfies`, where $id_P$ was introduced in $\tau$. The second group of facts in (10)–(12) describes the transitive closure $\mathscr{C}$, the existing installation $\mathscr{O}$, and latest versions of packages in $\mathscr{C}$ via the predicates `unit`, `installed`, and `newestversion`. Moreover, facts over the predicate `criterion` in (13) represent objectives $\#_i[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i$ occurring in $\mathbf{O}$ by an associated constant $c_{[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i}$ and the polarity $\#_i \in \{+,-\}$ along with the position $i$ in $\mathbf{O}$. E.g., the facts obtained for the CUDF document in Figure 2 and the sequence $\mathbf{O} = (-\mathbf{C}^{\mathscr{P}}_{\mathscr{O}}, -\mathbf{D}^{\mathscr{P}}_{\mathscr{O}})$ of objectives are shown in Figure 4. Note that, in view of unspecified objectives regarding recommendations, the respective interdependency of package (`dep`, 3) is not reflected in the facts. However, when $-\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}$ would be added to $\mathbf{O}$, '`recommends(dep,3,`$id_{\{(\mathtt{recomm},1)\}}$`,1).`' along with further facts describing (`recomm`, 1) (then also included in $\mathscr{C}$) would be obtained in $\pi$.

## 3 Grounding and Solving

The facts $\pi$ generated by the preprocessor serve as problem-specific input to the ASP components of *aspcud*, viz., the grounder *gringo* [7] and the solver *clasp* [8], while general knowledge about package configuration problems is provided via encodings. For one, the encoding `configuration.lp` in Figure 5 specifies admissible follow-up installations $\mathscr{P}$; for another, `optimization.lp` in Figure 6 encodes optimization criteria (violations) and corresponding penalties. The encodings are written in the first-order input language of *gringo*, which instantiates the contained variables wrt. $\pi$ to produce a propositional representation suitable for *clasp*. For space reasons, we confine the presentation to the encodings that appeared to be most successful in our preliminary, systematic experiments and are thus used by default in *aspcud*. However, major strengths of ASP are its first-order input language and the availability of grounders to instantiate them; this enables rapid prototyping of alternative problem formulations, and we indeed tested several encoding variants before deciding for the ones provided next.

## 3.1 Hard Constraints

Hard requirements for follow-up installations $\mathscr{P}$ are encoded in `configuration.lp`. Here, the rules in Line 3–10 are used to abstract from versions if a property applies to all (installable) versions of a package. Note that variables are universally quantified, where `P` stands for the name a package, `X` for a version of `P`, and `D` is an identifier, $id_P$, for a set $P$ of packages. In view of this, the auxiliary predicate `pconflict` defined in Line 3 projects out versions `X` from facts over `conflict` in $\pi$. The rule in Line 4 then lifts a conflict between some version of `P` (and packages fulfilling `D`) to the package name `P`, provided that all (installable) versions `X` conflict with `D`; in fact, the condition '`conflict(P,X,D) : unit(P,X)`', evaluated wrt. values for `P` and `D` given through `pconflict(P,D)`, refers to the conjunction of `conflict(P,X,D)` over all instances of `X` such that `unit(P,X)` holds. From the facts $\pi$ in Figure 4, `conflict(conf,`$id_{\{(\mathtt{feat},1)\}}$`)` and `conflict(feat,`$id_{\{(\mathtt{conf},2)\}}$`)` are derived via instances of the rules in Line 3 and 4, as `conflict(conf,2,`$id_{\{(\mathtt{feat},1)\}}$`)` and `conflict(feat,1,`$id_{\{(\mathtt{conf},2)\}}$`)` are provided by facts for the only (installable) versions `2` and `1` of `conf` and `feat`, respectively. The same approach to lift properties to package names `P` is applied to dependencies and satisfaction relationships (i.e., membership in a set $P$ referred to by some $id_P$, given via facts over the predicate `satisfies`).

While the rules described so far derive deterministic properties from facts, the "choice" rule in Line 14 of `configuration.lp` allows for guessing a follow-up installation $\mathscr{P}$. It describes that, for any instance of `(P,X)` specified by the predicate `unit`, one may freely choose whether to include `in(P,X)` in an answer set; and a follow-up installation $\mathscr{P}$ is given by the instances of `in(P,X)` belonging to an answer set. Hence, the rule in Line 14 opens up the candidate space for $\mathscr{P}$, which is however limited to the transitive closure $\mathscr{C}$ (determined via Algorithm 1) because facts over `unit` do not include packages outside $\mathscr{C}$. The rule in Line 15 again abstracts from the version `X` of a package `P` in $\mathscr{P}$ by projecting out `X` from `in(P,X)`. Once guessed, it remains to check whether a follow-up installation $\mathscr{P}$ is admissible. To this end, the rules in Line 17–24 collect the identifiers $id_P$ of target sets $P$ of package interdependencies, divided by `forbidden` and `requested` target sets in view of conflicts and dependencies, respectively, of packages in $\mathscr{P}$, and `satisfied` target sets are determined in turn. The actual checks are implemented via the "constraints" in Line 26–28, which deny follow-up installations $\mathscr{P}$ such that the target set of a `request` (due to some `install` or `upgrade` statement in the original CUDF document) or a `requested` package dependency is not `satisfied`; furthermore, a target set `forbidden` in view of some conflict must not be `satisfied`. For instance, the requirement expressed by '`request(`$id_{\{(\mathtt{inst},3),(\mathtt{inst},2),(\mathtt{inst},1)\}}$`).`' in Figure 4 along with the constraint in Line 26 deny follow-up installations $\mathscr{P}$ that do not include any of the packages `(inst,3)`, `(inst,2)`, and `(inst,1)` because `satisfied(`$id_{\{(\mathtt{inst},3),(\mathtt{inst},2),(\mathtt{inst},1)\}}$`)` can be derived only if `in(inst,`$n$`)` holds for some $n \in \{1,2,3\}$. If so, an instance of the rule in Line 23 as well as the rules in Line 15 and 24 apply, where the latter relies on `satisfies(inst,`$id_{\{(\mathtt{inst},3),(\mathtt{inst},2),(\mathtt{inst},1)\}}$`)`, which abstracts from versions of `inst`. Note that such abstractions and the rules in Line 18, 21, and 24 exploiting them are in principle redundant, since analogous rules considering versions in Line 17, 20, and 23 achieve the same effect, once a version `X` of `P` is determined via `in(P,X)`. However, our preliminary empirical comparisons between several encoding variants suggested `configuration.lp` in Figure 5 as the most "efficient" encoding. Finally, an admissible follow-up installation $\mathscr{P}$ can be read off from instances of `in(P,X)` belonging to an answer set, and so we confine its displayed part accordingly in Line 32.

```
% analyze package interdependencies

 pconflict(P,D) :-   conflict(P,X,D).
  conflict(P,D) :-  pconflict(P,  D),  conflict(P,X,D) : unit(P,X).

  pdepends(P,D) :-    depends(P,X,D).
   depends(P,D) :-   pdepends(P,  D),   depends(P,X,D) : unit(P,X).

psatisfies(P,D) :-  satisfies(P,X,D).
 satisfies(P,D) :- psatisfies(P,  D), satisfies(P,X,D) : unit(P,X).

% generate follow-up installation

{ in(P,X) }     :- unit(P,X).
  in(P)         :-   in(P,X).

forbidden(D)    :-   in(P,X),  conflict(P,X,D).
forbidden(D)    :-   in(P),    conflict(P,  D).

requested(D)    :-   in(P,X),   depends(P,X,D).
requested(D)    :-   in(P),     depends(P,  D).

satisfied(D)    :-   in(P,X), satisfies(P,X,D).
satisfied(D)    :-   in(P),   satisfies(P,  D).

 :-   request(D), not satisfied(D).
 :- requested(D), not satisfied(D).
 :- forbidden(D),     satisfied(D).

% project output

#hide.  #show in/2.
```

**Figure 5:** ASP encoding of follow-up installations $\mathscr{P}$ wrt. facts $\pi$ (`configuration.lp`).

## 3.2 Soft Constraints

The encoding `optimization.lp` in Figure 6 builds on top of facts $\pi$ and `configuration.lp` to identify optimization criteria violations and to assign corresponding penalties. While the rule in Line 1 merely projects out versions `X` of packages `P` `installed` in $\mathscr{O}$, the rules in Line 5–12 recognize changes, additions, and removals of packages `P` in the transition from $\mathscr{O}$ to $\mathscr{P}$. Note that any such `violated` maintenance condition is considered only if associated objectives are specified via facts over the predicate `criterion` in $\pi$; for the facts in Figure 4, the rules in Line 5–8 and 11–12 of Figure 6 are applicable, given that the sequence $\mathbf{O} = (-\mathbf{C}^{\mathscr{P}}_{\mathscr{O}}, -\mathbf{D}^{\mathscr{P}}_{\mathscr{O}})$ of objectives is expressed via '`criterion(change,-1).`' and '`criterion(remove,-2).`' Objectives regarding latest versions of packages in $\mathscr{P}$ and recommendations are addressed by the rules in Line 13–14 and 15–16, respectively. Note that the latter uses a different format, `r(P,X,D)`, to indicate an unserved recommendation `D` of a package `P` in version `X`, where `D` is an identifier of the form $id_P$ for a target set $P$; in addition, the multiplicity of recommendation targets served by $P$ is given in `R`. (Since violations of the other optimization criteria, identified in Line 5–14, are counted once per package name `P`, their corresponding instances of `violated(C,P,1)`

```
 1  installed(P) :- installed(P,X).

 3  % identify optimization criteria violations

 5  violated(change,     P,      1) :- criterion(change,    L),
 6                                 installed(P,X), not in(P,X).
 7  violated(change,     P,      1) :- criterion(change,    L),
 8                             not installed(P,X),     in(P,X).
 9  violated(newpackage, P,      1) :- criterion(newpackage,L),
10                             not installed(P),       in(P).
11  violated(remove,     P,      1) :- criterion(remove,    L),
12                                 installed(P),   not in(P).
13  violated(uptodate,   P,      1) :- criterion(uptodate,  L),
14                             newestversion(P,X), not in(P,X), in(P).
15  violated(recommend,r(P,X,D),R) :- criterion(recommend, L),
16                                recommends(P,X,D,R), in(P,X), not satisfied(D).

18  % post optimization criteria

20  #minimize[ violated(C,P,W) = W @ -L : criterion(C,L) : L < 0 ].
21  #maximize[ violated(C,P,W) = W @  L : criterion(C,L) : L > 0 ].
```

**Figure 6:** ASP encoding of optimization criteria wrt. follow-up installations $\mathscr{P}$ (`optimization.lp`).

use `1` as default weight.) The **`#minimize`** and **`#maximize`** statements in Line 20 and 21 associate penalties (or rewards) with violations of objectives of the form $\#_i[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i$ in a sequence $\mathbf{O}$, reflected in $\pi$ by including '`criterion(`$c_{[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i}$`,`$\#_i i$`).`' (where $c_{[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i} \in \{$`newpackage`, `remove`, `change`, `uptodate`, `recommend`$\}$ and $\#_i \in \{+,-\}$). Instances of `violated(`$c_{[\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}]_i}$`,P,W)` in an answer set, derived via the rules in Line 5–16, are then penalized (or rewarded) with priority $i$ and weight `W`. Note that summation-based minimization applies (in Line 20) if $\#_i = -$ or maximization (in Line 21) if $\#_i = +$, while a later position $i$ in $\mathbf{O}$ indicates greater significance than preceding ones. For instance, the sequence represented by '`criterion(change,-1).`' and '`criterion(remove,-2).`' gives preference to the minimization of $\mathbf{D}^{\mathscr{P}}_{\mathscr{O}}$ and then considers the cardinality of $\mathbf{C}^{\mathscr{P}}_{\mathscr{O}}$ for breaking ties. As already mentioned, maximization objectives of the form $+\mathbf{O}^{\mathscr{P}}_{\mathscr{O}/\mathscr{U}}$ (aiming at many differences between $\mathscr{O}$ and $\mathscr{P}$, outdated packages in $\mathscr{P}$, or recommendations ignored by $\mathscr{P}$, respectively) seem of little practical use. Since they would still be allowed in the *user* track of competitions by mancoosi, the **`#maximize`** statement in Line 21 of Figure 6 is included to handle them.

The instantiation of `configuration.lp` and `optimization.lp` wrt. facts $\pi$, produced by *gringo*, is passed on to the ASP solver *clasp*, which searches for (optimal) answer sets of propositional logic programs. In the context of Linux package configuration, the major challenge lies in the optimization of objectives, given that available distributions are large and plenty installations are admissible (even when the transitive closure $\mathscr{C}$ is used to limit the scope of a follow-up installation $\mathscr{P}$). In view of this, we recently extended *clasp* by dedicated search strategies and heuristics for effective multi-criteria optimization [5]; by default, *aspcud* configures them by supplying the command line options `--opt-hierarch=1` and `--opt-heuristic=1` to *clasp*. (Default *clasp* options can be overridden via *aspcud* switch '`-c`'.) In a nutshell, these options instruct *clasp* to optimize multiple objectives successively in the order of significance by progressively improving objective values of answer sets until the problem of finding a better answer set turns out to be unsatisfiable, in which case optimization proceeds with the next (less significant) criterion. Further search parameters of *clasp* are, by default, set by supplying the command line options `--sat-prepro`, `--heuristic=vsids`, `--solution-recording`, `--restarts=128`, and `--local-restarts`. We determined the *clasp* setting utilized by *aspcud* via systematic experiments (see [5, 6] for an empirical comparison between *clasp* settings), and the successful participations of *aspcud* in recent trial-runs of the competition by mancoosi [13] were largely owed to the search capacities of *aspcud*'s solving component.

## 4 Experiments

The workflow of *aspcud* includes the steps of preprocessing, grounding, and solving (as well as converting an answer set representing a follow-up installation back to CUDF). Since *clasp* settings were already evaluated in [5, 6], the experiments presented here concentrate on the impact of preprocessing on residual problem size and its effect on solving efficiency. To be more precise, we compare problem size and search statistics wrt. ASP facts limited to the transitive closure $\mathscr{C}$ determined via Algorithm 1 against facts describing the whole universe $\mathscr{U}$ of packages (except for those that must not be installed in view of `remove` and `upgrade` requests).

Our experiments consider four benchmark classes, in the following referred to by *easy*, *difficult*, *impossible*, and *debian-dudf*, from the 2010 MISC competition by mancoosi [13]. Furthermore, we apply the sequences $(-\mathbf{C}^{\mathscr{P}}_{\mathscr{O}}, -\mathbf{D}^{\mathscr{P}}_{\mathscr{O}})$ and $(-\mathbf{N}^{\mathscr{P}}_{\mathscr{O}}, -\mathbf{R}^{\mathscr{P}}_{\mathscr{U}}, -\mathbf{U}^{\mathscr{P}}_{\mathscr{U}}, -\mathbf{D}^{\mathscr{P}}_{\mathscr{O}})$ of objectives (in increasing order of significance) used in the tracks called *paranoid* and *trendy*. (Arbitrary sequences of objectives can be provided as arguments to *aspcud*, as required in the *user* track.) Note that, although the instances are the same in *paranoid* and *trendy* mode, optimization wrt. the latter is usually more difficult in view of more criteria. We ran the experiments under MISC conditions, imposing a time limit of 300 seconds, on an Intel Xeon E5520 machine, equipped with 2.27GHz processors and 48GB main memory, under Linux.

Table 1 summarizes experimental results, separately for *paranoid* and *trendy* objectives, where the first two columns provide the considered benchmark class along with the number $n$ of its instances. The entries in the other columns contrast statistics obtained with transitive closure computation (before '/') against the ones obtained without it (after '/'). Average problem sizes in terms of number of variables and constraints, as reported by *clasp*, are provided in the third and fourth column. The fifth column gives average solving times, with timeouts (in parentheses) taken as 300 seconds. The numbers of choices, conflicts, and answer sets (including intermediate ones) reported by *clasp* are shown in the last three columns, here averaging over the instances finished within the time limit in both preprocessing modes.

With transitive closure computation enabled, we observe a reduction of both variables and constraints by about one order of magnitude (a bit less on the *debian-dudf* class). This can be explained by the fact that typical installations include only a fraction of the available packages. Furthermore, the reductions in size are greater wrt. *paranoid* objectives because they disregard recommendations, which are considered in *trendy* mode. The solving times also reduce by one order of magnitude for *paranoid*, yet less for the more difficult problems solved in *trendy* mode; however, eight more instances are solved in time with transitive closure computation enabled. Interestingly, the numbers of conflicts and answer sets (taken only over instances that did not time out) are comparable. This indicates that *clasp*'s optimization approach is able to focus on relevant problem parts, even without a priori limitation to the transitive closure. Nonetheless, the numbers of choices are much greater (again an order of magnitude) for whole package universes, providing a clear indication of the benefits of limiting the scope of follow-up installations. In fact, even when unnecessary variables and constraints do not render a problem more difficult, the solving time suffers from additional efforts spent on assigning the variables and testing the constraints.

| *paranoid* | *n* | variables | constraints | time (t/o) | choices | conflicts | answer sets |
|---|---|---|---|---|---|---|---|
| *easy* | 20 | 6K/ 69K | 6K/ 91K | 1( 0)/ 9( 0) | 35K/ 1,932K | 22/ 27 | 66/192 |
| *difficult* | 22 | 11K/158K | 10K/180K | 2( 0)/ 25( 0) | 42K/ 717K | 5K/ 4K | 67/87 |
| *impossible* | 14 | 36K/404K | 64K/654K | 6( 0)/ 98( 0) | 90K/ 992K | 7K/ 5K | 58/81 |
| *debian-dudf* | 18 | 40K/189K | 82K/359K | 6( 0)/ 40( 0) | 232K/ 953K | 2K/ 1K | 220/116 |
| *trendy* | *n* | variables | constraints | time (t/o) | choices | conflicts | answer sets |
| *easy* | 20 | 9K/ 80K | 11K/121K | 1( 0)/ 14( 0) | 117K/ 3,690K | 1K/ 2K | 203/341 |
| *difficult* | 22 | 21K/175K | 26K/232K | 155(11)/196(12) | 279K/ 3,057K | 26K/28K | 270/400 |
| *impossible* | 14 | 70K/438K | 136K/782K | 163( 6)/259(12) | 462K/ 2,949K | 12K/12K | 289/253 |
| *debian-dudf* | 18 | 51K/207K | 111K/432K | 20( 0)/106( 1) | 946K/10,910K | 35K/51K | 678/874 |

**Table 1:** Experiments assessing the impact of preprocessing via Algorithm 1 on *aspcud*'s performance.

## 5 Discussion

We presented the workflow of the ASP-based Linux package configuration tool *aspcud*. In particular, we detailed the preprocessing applied to convert CUDF input to ASP facts suitable for the ASP components of *aspcud*. Related approaches rely on conversions from CUDF to Integer Linear Programming [14], Maximum Satisfiability [9], or Pseudo-Boolean Optimization [1]. Although all conversions, including ours, closely follow the specification of CUDF [17] and differ primarily in their target formats, there still are some differences that deserve attention. Unlike other package configuration tools, *aspcud* compiles CUDF input into ASP facts, while constraints as well as optimization criteria on follow-up installations are provided separately via general problem encodings. In fact, *aspcud* is equipped with several encoding variants (selectable via switch '`-e`'), although we here only detailed the most promising variants according to our empirical investigations. For another, the preprocessors of package configuration tools trace indirections in view of arithmetic expressions (over versions), package formulae, and virtual packages admitted in CUDF back to the (installable) packages underneath. In our ASP fact format (cf. Figure 3), we however associate target sets $P$ of package interdependencies with identifiers $id_P$ in order to avoid unfolding steps upon fact generation. To our knowledge, the preprocessors of other package configuration tools perform such unfolding, and it is an interesting (unresolved) question whether structural entities of the form $id_P$ are rather beneficial or a handicap for search. Regarding modeling in ASP (cf. Figure 5 and 6), the consequent usage of identifiers $id_P$ helped to keep the encodings concise and thus easy to maintain and modify. Despite of the different input formats used in ASP and the solving components of other package configuration tools, the principal approach of *aspcud*'s preprocessor to limit the scope of follow-up installations is independent of back-end solvers; however, an additional "constraint formulator" would be required for back-ends lacking general-purpose grounders. Concerning subjects to future investigation, we speculate that further improvements of problem encodings or the exploration of characteristic structures in Linux distributions (if any) might boost the performance of package configuration tools, in addition to ongoing enhancements of their search engines.

**Acknowledgments.** This work was partly funded by DFG grant SCHA 550/8-2. We are grateful to Daniel Le Berre for useful discussions on the topic, to the mancoosi project team for organizing MISC, and to the anonymous referees for helpful comments.

# References

[1] J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva & P. Rapicault (2010): *Solving Linux Upgradeability Problems Using Boolean Optimization*. In Lynce & Treinen [12], pp. 11–22, doi:10.4204/EPTCS.29.2.

[2] *aspcud*. `http://www.cs.uni-potsdam.de/wv/aspcud`.

[3] C. Baral (2003): *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, doi:10.1017/CBO9780511543357.

[4] A. Biere, M. Heule, H. van Maaren & T. Walsh, editors (2009): *Handbook of Satisfiability*. IOS Press.

[5] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2011): *Multi-Criteria Optimization in Answer Set Programming*. In J. Gallagher & M. Gelfond, editors: *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, Leibniz International Proceedings in Informatics, pp. 1–10, doi:10.4230/LIPIcs.ICLP.2011.1.

[6] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2011): *Multi-Criteria Optimization in ASP and its Application to Linux Package Configuration*. In Le Berre & Van Gelder [10]. To appear.

[7] M. Gebser, R. Kaminski, A. König & T. Schaub (2011): *Advances in Gringo Series 3*. In J. Delgrande & W. Faber, editors: *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, Springer, pp. 345–351, doi:10.1007/978-3-642-20895-9_39.

[8] M. Gebser, B. Kaufmann, A. Neumann & T. Schaub (2007): *Conflict-Driven Answer Set Solving*. In M. Veloso, editor: *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, AAAI Press/The MIT Press, pp. 386–392.

[9] M. Janota, I. Lynce, J. Marques-Silva & V. Manquinho (2011): *PackUp: Tools for Package Upgradability Solving*. In Le Berre & Van Gelder [10]. To appear.

[10] D. Le Berre & A. Van Gelder, editors (2011): *Proceedings of the Second Workshop on Pragmatics of SAT (PoS'11)*. To appear.

[11] V. Lifschitz, F. van Harmelen & B. Porter, editors (2008): *Handbook of Knowledge Representation*. Elsevier Science.

[12] I. Lynce & R. Treinen, editors (2010): *Proceedings of the First International Workshop on Logics for Component Configuration (LoCoCo'10)*. *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 29, doi:10.4204/EPTCS.29.

[13] mancoosi — *managing software complexity*. `http://www.mancoosi.org`.

[14] C. Michel & M. Rueher (2010): *Handling Software Upgradeability Problems with MILP Solvers*. In Lynce & Treinen [12], pp. 1–10, doi:10.4204/EPTCS.29.1.

[15] *potassco*. `http://potassco.sourceforge.net`.

[16] T. Syrjänen (2000): *Including Diagnostic Information in Configuration Models*. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv & P. Stuckey, editors: *Proceedings of the First International Conference on Computational Logic (CL'00)*, Springer, pp. 837–851, doi:10.1007/3-540-44957-4_56.

[17] R. Treinen & S. Zacchiroli (2009): *Common Upgradability Description Format (CUDF) 2.0*. Technical Report 003, [13].