A short guide to **user preferences** for the MISC dependency solvers and a proposal for its extension*

Roberto Di Cosmo
Univ Paris Diderot, Sorbonne Paris Cité,
PPS, UMR 7126, CNRS, F-75205 Paris, France
roberto@dicosmo.org

2014-07-24 - Revision : dbb706e

1 The basics

When performing an installation or an upgrade, the user of a package manager explicitly formulates a *user request* mentioning the packages that need to be installed or upgraded, possibly with same constraints (e.g. to install package `foo` in some version greater then `3.2-5`).

The user may also be concerned about the packages that are changed, the packages that are not up to date, the packages that get removed, or even more exotic features like "the overall installed size" of the solution proposed by the package manager. This is possible when using package managers designed according to the principles stated in [1], and using the dependency solvers that participated in the latest MISC competition [1].

These solvers are all able to read the CUDF format [7] and provide a specialised *preference language* that allows to express these high-level concerns. A *preference expression* is built from four basic ingredients:

package selectors a predefined set of keywords allow to identify in a proposed solution certain classes of packages (the ones that changed, the ones that got removed, etc.)

measurements a predefined set of measurement functions can be appliead to a package selector to obtain an integer value (the number of package selected, the number of packages selected that are up-to-date, etc.)

maximisation/minimisation one can ask the solver to find a solution that maximises or minimises the value of a measurement

aggregation one can ask the solver to aggregate several maximised or minimised measurements in lexicographical order

We summarise in what follows the package selectors and the measurements available in the latest preference language.

2 Package selectors

In the following we describe the available *package selectors*. Notice that in what follows S stands for the solution proposed by the solver, I stands for the packages installed before calling the solver,

*Work performed at IRILL `http://www.irill.org/`, center for research and innovation on Free Software.
[1] http://www.mancoosi.org/misc/

IntReq stands for the part of the user request specifying the packages to install, and *UpgReq* stands for the part of the user request specifying the packages to upgrade.

solution denotes the set S of packages that are in the solution found by the solver

changed denotes $\{p \in S \mid p \notin I\} \cup \{p \in I \mid p \notin S\}$, in other words the symmetric set difference between S and I. That is, changed denotes the set of packages that have been newly installed or removed. Note that when a package with name foo gets upgraded from version 1 to version 2 this means that (foo,1) gets removed and (foo,2) gets installed, and as a consequence the set changed contains both packages (foo,1) and (foo,2).

new denotes $\{p \in S \mid p.name \notin I.name\}$, the set of packages that are finally installed, and for which no package of the same name was initially installed.

removed denotes $\{p \in I \mid p.name \notin S.name\}$, the set of packages that were initially installed, and for which no package of the same name is finally installed.

up denotes $\{p \in S \mid \exists q \in I$ with $q.name = p.name$, and $\forall q \in I$ with $q.name = p.name : q.version < p.version\}$. In other words, up is the set of upgraded packages. In case multiple versions of packages with the same name may be installed, the upgrade consists of the packages with a version higher than any previously installed version.

down denotes $\{p \in S \mid \exists q \in I$ with $q.name = p.name$, and $\forall q \in I$ with $q.name = p.name : q.version > p.version\}$. In other words, up is the set of downgraded packages. In case multiple versions of packages with the same name may be installed, the downgrade consists of the packages with a version higher than any previously installed version.

Starting from version `1.9.1`, the `aspcud` solver [5] also supports the following additional selectors

installrequest denotes $\{p \in S \mid p$ mentioned in $InstReq\}$, i.e. the packages in the solution that the user explicitly requested to install

upgraderequest denotes $\{p \in S \mid p$ mentioned in $UpgReq\}$, i.e. the packages in the solution that the user explicitly requested to upgrade

request is the union of the above two

3 Measurements

On each package selector X, we can ask the solver to compute one of the following measures:

count(X) is the number of packages in X

sum(X,f) is the sum over all packages in the set X of the value of their *integer* field f. A typical example of this would be `sum(solution,size)`, with `size` an integer-valued property of packages indicating their size.

notuptodate(X) is the number of packages in X that are not in the latest known version.

unsat_recommends(X) counts the number of disjunctions in Recommends-fields of packages in X that are not satisfied by S.

aligned(X,g1,g2) is formally $card(\{(x.g1, x.g2) \mid x \in X\}) - card(\{x.g1 \mid x \in X\})$. In other words, we first cluster the packages in X according to their values at the properties g1 and g2 and count the number of clusters, yielding a value $v1$. Then we do the same when clustering only by the property g1, yielding a value $v2$. The value returned is then $v1 - v2$. This measure can be used to count *version changes* as defined in [4].

4 Optimising and combining measurements

The measures described above are used as functions that the solver is then asked to maximise or minimise. In the preference language, maximisation is specified by prepending the plus sign + to a measure, and minimisation is specified by prepending a minus sign − to a measure.

For example, `-count(changed)` requires a solution that minimises changes with respect to the initial configuration, while `+count(up)` requests a solution maximising the upgrades.

Criteria can be combined in lexicographical order; for example `-count(removed),-count(changed)` is a preference expression that specifies a solution which minimises the number of removed packages, and then minimises the number of changed ones.

5 Examples of usage

5.1 Examples preferences when installing your packages

Simple. Here are two preferences that roughly correspond to a *paranoid* user which is interested in installing a package with minimal changes to the system, and to a *trendy* user that wants to get the latest and best version of all current packages. They both want to avoid removal of installed packages, but share little less.

paranoid `-count(removed),-count(changed)`

trendy `-count(removed),-notuptodate(solution),-count(new)`

More refined. Using the selectors `request` and `down`, it is possible to write a more refined version of the above preferences, that may correspond to the needs of both categories of users: here we ask the solver to get the latest version only for the packages explicitly mentioned in the request.

paranoidtrendy `-count(removed),-notuptodate(request),-count(down),-count(changed)`

Repairing a broken system configuration. If a system has got into an inconsistent state (package dependencies are broken, conflicts creeped in), it is quite simple to repair it by issueing an empty request with one of the following user preferences

fixup simple `-count(changed)`

fixup trendy `-count(changed),-count(down),-notuptodate(solution)`

More exotic examples. It is also possible to exploit user preferences to obtain a bill of materials to build a system with specific properties, like a minimal disk footprint (as is done in `https://github.com/rdicosmo/ami-thinner` to create minimal Debian virtual machines), or the largest set of compatible components.

Minimal system size `-sum(solution,installedsize),-count(solution)`

Noah's ark `+count(solution)`

Noah's ark, fresh `-notuptodate(solution),+count(solution)`

6 Mixing user request and solver preferences for expressivity

There may be situations where the precise objective the user is pursuing can be described by using a combination of the user requestand the solver preferences.

A typical example is the removal of *cruft*, that can accumulate on a system in the form of packages that ended up installed at some point, to satisfy dependencies, but that are now no longer needed by any of the packages explicitly installed by the user (often called *root* packages). This functionality is offered for example by package managers like `apt-get` or `aptitude` that provide an *autoremove* command.

A package manager built on top of CUDF-compatible dependency solvers can provide the same functionality by simply specifying in an *install* user request all the *root* packages, at their current version, and then using the preference expression `-count(solution)`. This will effectively ensure that all root packages stay installed at their current version, and then minimise the number of extra packages present.

To get at the same time the latest version of the root packages, one can use an *upgrade* user request mentioning all the root packages (that will prevent downgrades of the root packages) and then use the preference expression `-notuptodate(request),-count(down),-count(solution)`. The resulting effect will be to pick a solution that has the latest version of the root packages, while minimising downgrades and irrelevant extra packages.

7 Older preference language

Solvers participating in earlier versions of the MISC competition supported a simpler preference language, that can be directly translated into the more sophisticated one described above.

The correspondence is given below.

Old language	New language
removed	count(removed)
new	count(new)
changed	count(changed)
notuptodate	notuptodate(solution)
unsat_recommends	unsat_recommends(solution)

8 Solvers supporting user preferences

At the time of this writing, here is the list of available dependency solvers supporting the different preference languages:

full selectors and measures
: `aspcud`[2] versions `1.9.1` and later

MISC 2012 selectors and measures
: `aspcud` in versions `1.8` and later;
: `p2cudf` [3]

[2] `http://sourceforge.net/projects/potassco/files/aspcud/`
[3] `http://wiki.eclipse.org/Equinox/p2/CUDFResolver`

older preference language
aspcud versions 1.9.1 and later,
aspcud versions before 1.8 (e.g. version 2011.03.17 in Debian and Ububntu);
mccs[4],
and packup[5].

8.1 Getting your external solver

The p2cudf solver is Java based, and sould be quite portable, even if it is not the fastest among the available solvers. For Debian and Ubuntu users, the mccs, packup and aspcud solvers are installable out of the box.

In case one needs a solver on a platform that does not easily accomodate it, either because a packages version is not available, or because the system has little resources (like Arduino or RaspberriPi boards), it is now possible to get a solver *in the cloud*. It is enough to go to http://cudf-solvers.irill.org and follow the instructions therein.

9 Package managers that can use external dependency solvers

These solvers can be used by recent versions of the apt-get and opam package managers.

9.1 Using external solvers with apt-get

Current versions of apt-get have an option --solver that allows to specify an external solver. Notice though that because of the inner working of apt-get, one gets better result by using the wrapper script apt-cudf-get that accepts the same options as apt-get.

9.2 Using external solvers with opam

Thanks to the work done by the Mancoosi team at Irill and OCamlPro, in the DORM project, the aspcud solver is supported out of the box in opam since 1.0. With a current version of opam, one can use any other CUDF compatible solver: typing opam --help shows

```
...
OPTIONS
  --criteria=CRITERIA
      Specify user preferences for dependency solving for this run.
      Overrides both $OPAMCRITERIA and $OPAMUPGRADECRITERIA.
      For details on the supported language, see
      http://opam.ocaml.org/doc/Specifying_Solver_Preferences.html.
      The default value is -removed,-notuptodate,-changed for upgrades,
      and -removed,-changed,-notuptodate otherwise.

  --cudf=FILENAME
      Debug option: Save the CUDF requests sent to the solver to
      FILENAME-<n>.cudf.

  --solver=CMD
      Specify the name of the external dependency solver. The default
      value is aspcud
...
```

[4] http://www.i3s.unice.fr/~cpjm/misc/mccs.html
[5] http://sat.inesc-id.pt/~mikolas/sw/packup/

10 Proposed extensions: filters, set operators, unsatclauses

Following recent discussions on package management issues, we propose to extend the preference language by introducing an additional package selector, `filter`, closing the package selectors by the union, intersection, and difference set operations, and generalising the `unsat_recommends`.

The intended syntax and semantics are described below:

Selectors

`filter(`*`field`* `op` *`value`*`)` denotes $\{p \in S \mid p.field\ op\ value\}$, the set of packages in the solution S whose field *field* contains a value v s.t. v *op value*. For example `filter(root = true)` will select all root packages in the solution (as seen above, in `opam`, that's the set of packages whose installation was explicitly requested by the user, as opposed to packages that are pulled in to satisfy dependencies).

set operators (and, or, minus) have the usual semantics as intersection, union and difference

With filters, it is pretty easy to rewrite the example given in Section 6 as the one liner

```
-count(changed  and  filter(root=true)),-count(solution)
```

And it is now possible to write a preference expression that roughly means "do not remove root packages, and bring them to their latest version"

```
-count(removed  and  filter(root=true)),-notuptodate(filter(root=true)
```

Measures

`unsatclauses(`X`,`*`name`*`)` counts the number of unsatisfied clauses in the field *name* of the packages in X. The current `unsat_recommends(`X`)` measure would then simply become a special case, and can be encoded as `unsatclauses(`X`,Recommends)`.

11 Conclusions and future work

User preferences are a powerful tool to orient a dependency solver towards solutions that correspond to high-level user desiderata. We have presented here the current state of the art for the class of solvers that did take part in the annual MISC competition over the past years.

More work may be needed to refine and expand the user preferences language, in order to improve its expressivity, according to concrete use cases, and a concrete proposition for an extension is presented in this document.

References

[1] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of System and Software Science*, 85(10):2228 – 2240, 2012. Automated Software Evolution.

[2] J. Argelich, D. L. Berre, I. Lynce, J. P. M. Silva, and P. Rapicault. Solving linux upgradeability problems using boolean optimization. In I. Lynce and R. Treinen, editors, *LoCoCo*, volume 29 of *EPTCS*, pages 11–22, 2010.

[3] D. L. Berre and A. Parrain. On sat technologies for dependency management and beyond. In S. Thiel and K. Pohl, editors, *SPLC (2)*, pages 197–200. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[4] R. Di Cosmo, O. Lhomme, and C. Michel. Aligning component upgrades. In C. Drescher, I. Lynce, and R. Treinen, editors, *Proceedings Second Workshop on Logics for Component Configuration*, volume 65, pages 1–11, 2011.

[5] M. Gebser, R. Kaminski, and T. Schaub. aspcud: A linux package configuration tool based on answer set programming. In C. Drescher, I. Lynce, and R. Treinen, editors, *LoCoCo*, volume 65 of *EPTCS*, pages 12–25, 2011.

[6] C. Michel and M. Rueher. Handling software upgradeability problems with milp solvers. In I. Lynce and R. Treinen, editors, *LoCoCo*, volume 29 of *EPTCS*, pages 1–10, 2010.

[7] R. Treinen and S. Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project, Nov. 2009. `http://www.mancoosi.org/reports/tr3.pdf`.

A BNF of the user preference language

The following BNF grammar describes the preference expression language. Notice that the green nonterminal `reqsel` describes the package selector extension implemented by `aspcud`, and the blue part of the `selector` and `bynarym` productions correspond to the proposed extensions proposed in section 10.

⟨prefexpr⟩ ⊨ ⟨optim⟩ | ⟨optim⟩ , ⟨prefexpr⟩
⟨optim⟩ ⊨ + ⟨measure⟩ | - ⟨measure⟩
⟨measure⟩ ⊨ ⟨unarym⟩ (⟨selector⟩) | ⟨bynarym⟩ (⟨selector⟩ , *field*) | aligned (⟨selector⟩ , *field* , *field*)
⟨unarym⟩ ⊨ count | notuptodate | unsat_recommends
⟨binarym⟩ ⊨ sum | unsatclauses
⟨selector⟩ ⊨ ⟨basicsel⟩ | ⟨reqsel⟩ | ⟨filtersel⟩ | (⟨selector⟩) | ⟨selector⟩ ⟨setop⟩ ⟨selector⟩
⟨basicsel⟩ ⊨ new | changed | removed | up | down | solution
⟨reqsel⟩ ⊨ request | installrequest | upgraderequest

⟨filtersel⟩ ⊨ filter (*fieldname* ⟨op⟩ *value*)
⟨op⟩ ⊨ `=` | `>` | `<` | `>=` | `<=` | `<>`
⟨setop⟩ ⊨ `and` | `or` | `minus`