The Dose-Debcheck Primer

Pietro Abate, Roberto Di Cosmo, Ralf Treinen, Stefano Zacchiroli

January 19, 2020

The dose-debcheck tool determines, for a set of package control stanzas, called *the repository*, whether packages of the repository can be installed relative to the repository. Typically, the repository is a `Packages` file of a Debian suite. The installability check is by default performed for all package stanzas in the repository, but may be also be restricted to a subset of these.

This primer applies to version 5.0.1 of dose-debcheck.

Contents

1 Input Data: Packages and Repositories

1.1 Packages

Debian control stanzas are defined in `deb-control (5)`. For dose-debcheck only the following fields are relevant, all others are ignored:

Package giving the package name. dose-debcheck is more liberal as to which package names are acceptable, for instance it allows a slightly larger character set than the debian policy for constituting names. Required.

Version giving the version of the package. The version must be in conformance with the Debian policy. Required.

Architecture specifying the architectures on which the package may be installed. Required.

Multiarch specifies whether the package may be installed simultaneously for different architectures, and whether it may satisfy dependencies across architecture boundaries. Values may be `No`, `Same`, `Foreign`, or `Allowed` ([**?**]). Optional, defaults to `No`.

Depends is a list of items required for the installation of this package. Each item is a package name optionally with a version constraint, or a disjunction of these. Items may also be annotated with `:any`. Optional, defaults to the empty list.

Pre-Depends are by dose-debcheck treated like Depends.

Conflicts is a list of package names, possibly with a version constraint, that cannot be installed together with the package. Optional, defaults to the empty list.

Breaks are by dose-debcheck treated like Conflicts.

Provides is a list of names symbolizing functionalities realized by the package. They have to be taken into account for dependencies and conflicts of other packages, see Section 2. Optional, defaults to the empty list.

Essential specifies whether the package must be installed (`yes` or `no`). Optional, defaults to `no`.

In particular, `Recommends` and `Suggests` are ignored by dose-debcheck. Also, dose-debcheck does not check for the presence of fields that are required by Debian policy but that are not relevant for the task of dose-debcheck, like `Maintainer` or `Description`.

Also note that dose-debcheck is slightly more liberal than the Debian policy in accepting input, and hence cannot be used to check strict policy conformance of package stanzas.

1.2 Repositories

A *repository* is a set of package stanzas. This set may be given to dose-debcheck in form of a single file or as several files, in the latter case the repository is constituted by all stanzas in all input files (see Section 3). dose-debcheck assures that the repositories has two important properties:

1. We assume that there are no two package stanzas in the repository that have the same values of all three fields `Package`, `Version`, and `Architecture`. Having different versions for the same package name is OK, as it is of course OK to have two stanzas with different package names and the same version. In other words, the dose-debcheck tool uses internally the triple of name, architecture and version as an identifier for packages.

   In the following, when we speak of *a package*, we mean a precise package stanza that is identified by a name, a version, and an architecture, like the package of name `gcc` in version `4:4.3.2-2` and for architecture `amd64`. The stanza with name `gcc` and version `4:4.4.4-2` for architecture `amd64` would constitute a different package.

   If the input contains several stanzas with the same name, version and architecture then all but the last such stanza are dropped, and a warning message is issued.

   Example: The following input does not constitute a repository:

   ```
   Package: abc
   Version: 42
   Architecture: amd64
   Depends: xyz

   Package: abc
   Version: 42
   Architecture: amd64
   Depends: pqr
   ```

   The reason is that the triple $(abc, 42, amd64)$ is not unique. dose-debcheck will warn us that it only accepts the second stanza and drops the first one from its input:

`(W)Debian: the input contains two packages with the same name, version and architect`

2. We assume that Multiarch information is consistent: If the repository contains packages with the same name and version and different architecture then both packages have to agree on the value of their `Multiarch` field.

2 Installability

2.1 A Precise Definition

In order to understand what installability exactly means for us we need a little bit of theory. Let R be a repository (see Section 1). An *R-installation set*, or sometimes simply called an *R-installation*, is a subset I of R that has the following four properties:

flatness: I does not contain two different packages with the same name (which then would have different versions or architecture), unless the package is marked as `Multiarch=Same`. If package (p, a, n) has `Multiarch=Same` then I just must not contain any package with name p and a version different from n.

abundance: For each package p in I, every of its dependencies is satisfied by some package q in I, either directly or through a virtual package in case the dependency does not carry a version constraint.

- If q has a Multiarch value of `No` or `Same` then the architecture of q must be the same as the architecture of p.
- If q has a Multiarch value of `Foreign` then the architecture of q may be different then the architecture of p.
- If q has a Multiarch value of `Allowed` then the architecture of q must be the same as the architecture of p, or the dependency relation must carry the annotation `:any`.

In this context, the architecture value `all` is identified with the native architecture [**?**].

peace: For each package in I and for each item in its list of conflicts, no package in I satisfies the description of that item. As an exception, it is allowed that a package in I both provides a virtual package and at the same time conflicts with it.

foundation: If package $(p, n) \in R$ is essential, then I must contain a package (p, m) such that (p, m) is essential.

Hence, the notion of an installation captures the idea that a certain set of packages may be installed together on a machine, following the semantics of binary package relations according to the Debian Policy. The foundation requirement

expresses that essential packages must be installed; it is formulated in a way that also caters to the (extremely rare) case that a package changes its `Essential` value between different versions. The foundation property may be switched off by giving the option `--deb-ignore-essential`.

Example: Let R be the following repository:

```
Package: a
Version: 1
Depends: b (>= 2) | v

Package: a
Version: 2
Depends: c (> 1)

Package: b
Version: 1
Conflicts: d

Package: c
Version: 3
Depends: d
Conflicts: v

Package: d
Version: 5
Provides: v
Conflicts: v
```

The following subsets of R are not R-installation sets:

- The complete set R since it is not flat (it contains two different packages with name a)
- The set $\{(a, 1), (c, 3)\}$ since it not abundant (the dependency of $(a, 1)$ is not satisfied, nor is the dependency of $(c, 3)$).
- The set $\{(a, 2), (c, 3), (d, 5)\}$ since it is not in peace (there is conflict between $(c, 3)$ and $(d, 5)$ via the virtual package v)

Examples of R-installation sets are

- The set $\{(d, 5)\}$ (self conflicts via virtual packages are ignored)
- The set $\{(a, 1), (b, 1)\}$
- The set $\{(a, 1), (d, 5)\}$

A package (p, n) is said to be *installable* in a repository R if there exists an R-installation set I that contains (p, n).

Example: In the above example, $(a, 1)$ is R-installable since it is contained in the R-installation set $\{(a, 1), (d, 5)\}$.

However, $(a, 2)$ is not R-installable: Any R-installation set containing $(a, 2)$ must also contain $(c, 3)$ (since it is the only package in R that can satisfy the dependency of $(a, 2)$ on $c(> 1)$, and in the same way it must also contain $(d, 5)$. However, this destroys the peace as $(c, 3)$ and $(d, 5)$ are in conflict. Hence, no such R-installation set can exist.

2.2 What Installability does Not Mean

- Installability in the sense of dose-debcheck only concerns the relations between different binary packages expressed in their respective control files. It does not mean that a package indeed installs cleanly in a particular environment since an installation attempt may still fail for different reasons, like failure of a maintainer script or attempting to hijack a file owned by another already installed package.

- Installability means theoretical existence of a solution. It does not mean that a package manager (like `aptitude`, `apt-get`) actually finds a way to install that package. This failure to find a solution may be due to an inherent incompleteness of the dependency resolution algorithm employed by the package manager, or may be due to user-defined preferences that exclude certain solutions.

2.3 Co-installability

One also should keep in mind that, even when two packages are R-installable, this does not necessarily mean that both packages can be installed *together*. A set P of packages is called *R-co-installable* when there exists a single R-installation set extending P.

Example: Again in the above example, both $(b, 1)$ and $(d, 5)$ are R-installable; however they are not R-co-installable.

See Section 6 on how co-installability can be encoded.

3 Invocation

3.1 Basic usage

dose-debcheck accepts several different options, and also arguments.

```
dose-debcheck [option] ... [file] ...
```

The package repository is partionend into a *background* and a *foreground*. The foreground contains the packages we are actually interested in, the background contains packages that are just available for satisfying dependencies, but for which we do not care about installability.

All arguments are interpreted as filenames of Packages input files, the contents of which go into the foreground. If no argument is given then metadata of foreground packages is read from standard input. In addition, one may specify listings of foreground packages with the option `--fg=<filename>`, and listings of background packages with the option `--bg=<filename>`. Input from files (but not from standard input) may be compressed with gzip or bzip2, provided dose-debcheck was compiled with support for these compression libraries.

The option `-f` and `-s` ask for a listing of uninstallable, resp. installable packages. The option `-e` asks for an explanation of each reported case. The exact effect of these options will be explained in Section 4.

Example: We may check whether packages in *non-free* are installable, where dependencies may be satisfied from *main* or *contrib*:

```
dose-distcheck  -f -e \
    --bg=/var/lib/apt/lists/ftp.fr.debian.org_debian_dists_sid_main_binary-amd64_Packages
    --bg=/var/lib/apt/lists/ftp.fr.debian.org_debian_dists_sid_contrib_binary-amd64_Packa
    /var/lib/apt/lists/ftp.fr.debian.org_debian_dists_sid_non-free_binary-amd64_Packages
```

3.2 Checking only selected packages

The initial distinction between foreground and background packages is modified when using the `--checkonly` option. This option takes as value a comma-separated list of package names, possibly qualified with a version constraint. The effect is that only packages that match one of these package names are kept in the foreground, all others are pushed into the background.

Example:

```
dose-debcheck --checkonly "libc6, 2ping (= 1.2.3-1)" Packages
```

3.3 Checking for co-installability

Co-installability of packages can be easily checked with the `--coinst` option. This option takes as argument a comma-separated list of packages, each of them possibly with a version constraint. In that case, dose-debcheck will check whether the packages specified are co-installable, that is whether it is possible to install these packages at the same time (see Section 2.3).

Note that it is possible that the name of a package, even when qualified with a version constraint, might be matched by several packages with different versions. In that case, co-installability will be checked for *each* combination of real packages that match the packages specified in the argument of the `--coinst` option.

Example: Consider the following repository (architectures are omitted for clarity):

```
Package: a
Version: 1
```

```
Package: a
Version: 2

Package: a
Version: 3

Package: b
Version: 10

Package: b
Version: 11

...
```

Executing the command `debcheck --coinst a (>1), b` on this repository will check co-installability of 4 pairs of packages: there are two packages that match `a (>1)`, namely package `a` in versions 2 and 3, and there are two packages that match `b`. Hence, the following four pairs of packages will be checked for co-installability:

1. (a,2), (b,10)
2. (a,2), (b,11)
3. (a,3), (b,10)
4. (a,3), (b,11)

Mathematically speaking, the set of checked tuples is the Cartesian product of the denotations of the single package specifications.

3.4 Changing the Notion of Installability

Some options affect the notion of installability:

- `--deb-ignore-essential` drops the Foundation requirement of installation sets (Section 2). In other words, it is no longer required that any installation set contains all essential packages.

Other options concern Multiarch:

- `--deb-native-arch=`a sets the native architecture to the value a. Note that the native architecture is not necessarily the architecture on which the tool is executed, it is just the primary architecture for which we are checking installability of packages. In particular, packages with the architecture field set to `all` are interpreted as packages of the native architecture [**?**].
- `--deb-foreign-archs=`$a_1, \ldots, a_n$ sets the foreign architectures to the list $a_1, \ldots, a_n$. Packages may only be installed when their architecture is the native architecture (including `all`), or one of the foreign architectures.

3.5 Filtering Packages and Multiarch

Filtering out packages is a different operation than pushing packages into the background (Section 3.2): Background packages are still available to satisfy dependencies, while filtering out a package makes it completely invisible.

- The effect of `--latest` is to keep only the latest version of any package.

3.6 Other Options

Other options controlling the output are explained in detail in Section 4. A complete listing of all options can be found in the dose-debcheck(1) manpage.

4 Output

The output of dose-debcheck is in the YAML format, see Section 6.2 for how to parse the output.

Without any particular options, dose-debcheck just reports some statistics:

Example:

```
% dose-debcheck rep1
background-packages: 0
foreground-packages: 4
total-packages: 4
broken-packages: 1
```

With the options `--failures` and `--successes`, dose-debcheck reports findings of the requested kind for all packages in the foreground. These options may be used alone or in combination. In any case, the status field tells whether the package is found to be installable (value `ok`) or non-installable (value `broken`).

Example:

```
% dose-debcheck --failures --successes rep1
report:
 -
  package: a
  version: 1
  architecture: amd64
  source: a (= 1)
  status: broken

 -
  package: a
  version: 2
  architecture: amd64
  source: a (= 2)
  status: ok
```

```
 -
  package: b
  version: 1
  architecture: amd64
  source: b (= 1)
  status: ok

 -
  package: c
  version: 3
  architecture: amd64
  source: c (= 3)
  status: ok


background-packages: 0
foreground-packages: 4
total-packages: 4
broken-packages: 1
```

With an additional `--explain` option, an explanation is given with each finding.

4.1 Understanding Explanations of Installability

An explanation of installability simply consists of an installation set in the sense of Section 2 containing the package in question.

Example:

```
% dose-debcheck --explain --successes rep1
report:
 -
  package: a
  version: 2
  architecture: amd64
  source: a (= 2)
  status: ok
  installationset:
   -
    package: c
    version: 3
    architecture: amd64
   -
    package: a
    version: 2
    architecture: amd64
```

```
-
 package: b
 version: 1
 architecture: amd64
 source: b (= 1)
 status: ok
 installationset:
  -
   package: b
   version: 1
   architecture: amd64
```

An installation set contains all essential packages (see Section 2), which may blow up the output of installability. Giving the option `--deb-ignore-essential` will avoid this, but will also alter the notion of installability in some corner cases (for instance, when a package needs a version of an essential package that is not available in the repository).

4.2 Understanding Explanations of Non-installability

Installability of a package is much easier to explain than non-installability. The reason for this is that in the former case we just have to give one installation that our tool has found, while in the latter case we have to explain why *all* possible attempts to install the package must fail. The first consequence of this observation is that the explanation in case of non-installability may consist of several components.

Example: Consider the following repository consisting of only two packages:

```
Package: a
Version: 1
Depends: b | c

Package: c
Version: 3
Conflicts: a
```

To explain why package (a,1) is not installable we have to say why all possible alternative ways to satisfy its dependency must fail:

- there is no package `b` in the repository
- the only version of package `c` in the repository is in conflict with package (a,1)

There may be several ways to satisfy dependencies due to alternatives in the dependencies in packages. Alternatives may occur in dependencies in different forms:

- explicitly, like in `Depends: b|c`,

- through dependency on a package that exists in several versions,
- through dependency on a virtual package which is provided by several (possibly versions of) real packages.

There is one component in the explanation for every possible way to choose among these alternatives in the dependencies.

There are only two possible reasons why an attempt to satisfy dependencies may fail:

1. dependency on a package that is missing from the repository,
2. dependency on a package that is in conflict with some other package we depend on (possibly through a chain of dependencies).

Each component of the explanation is either a missing package, or a conflict.

4.2.1 Explanation in Case of a Missing Package

A component of the explanation that corresponds to the case of a missing package consist of two stanzas:

- a `pkg` stanza that states the package that cannot satisfy one of its direct dependencies
- a `depchains` stanza containing the dependency chain that leads from the package we have found non-installable to the one that cannot satisfy its direct dependency.

Example: An explanation might look like this:

```
package: libgnuradio-dev
version: 3.2.2.dfsg-1
architecture: all
source: gnuradio (= 3.2.2.dfsg-1)
status: broken
reasons:
   -
    missing:
     pkg:
      package: libgruel0
      version: 3.2.2.dfsg-1+b1
      architecture: amd64
      unsat-dependency: libboost-thread1.40.0 (>= 1.40.0-1)
     depchains:
      -
       depchain:
        -
         package: libgnuradio-dev
```

```
 version: 3.2.2.dfsg-1
 Architecture: all
 Depends: libgnuradio (= 3.2.2.dfsg-1)
-
 package: libgnuradio
 ersion: 3.2.2.dfsg-1
 architecture: all
 depends: libgnuradio-core0
-
 package: libgnuradio-core0
 version: 3.2.2.dfsg-1+b1
 architecture: amd64
 depends: libgruel0 (= 3.2.2.dfsg-1+b1)
```

This tells us that `libgnuradio-dev` in version $3.2.2.dfsg-1$ is not installable, due to the fact that package `libgruel0` in version $3.2.2.dfsg-1+b1$ has a dependency `libboost-thread1.40.0 (>= 1.40.0-1)` that is not matched by any package in the repository. The dependency chain tells why package `libgnuradio-dev` in the given version might want to install `libgruel0`.

The depchains component gives all possible dependency chains (*depchains*, for short) from the root package (`libgnuradio-dev` in the above example) to the one where a direct dependency is not matched by any package (`libgruel0` in the example). We do not include the last node in the dependency chain to avoid a useless repetition.

In general there may be more then one path to reach a certain package from a given root package, in that case dose-debcheck will unroll all of them.

Example: In the following repository, package `a` is not installable since the dependency of package `d` cannot be satisfied:

```
Package: a
Architecture: amd64
Version: 1
Depends: b|c

Package: b
Architecture: amd64
Version: 1
Depends: d

Package: c
Architecture: amd64
Version: 3
Depends: d

Package: d
Architecture: amd64
Version: 42
```

```
Depends: x
```

There are two different ways how `a` arrives at a dependency on `d`. dose-debcheck reports the problem once, but lists the two paths from `a` to `d`:

```
% dose-debcheck -e -f --checkonly a rep1
report:
 -
  package: a
  version: 1
  architecture: amd64
  source: a (= 1)
  status: broken
  reasons:
   -
    missing:
     pkg:
      package: d
      version: 42
      architecture: amd64
      unsat-dependency: x
     depchains:
      -
       depchain:
        -
         package: a
         version: 1
         architecture: amd64
         depends: b | c
        -
         package: b
         version: 1
         architecture: amd64
         depends: d
      -
       depchain:
        -
         package: a
         version: 1
         architecture: amd64
         depends: b | c
        -
         package: c
         version: 3
         architecture: amd64
         depends: d
```

4.2.2 Explanation in Case of a Conflict

The other possible cause of a problem is a conflict. In that case, the explanation consists of a `conflict` stanza giving the two packages that are in direct conflict with each other. Next, we have two `depchain` stanzas that lead to the first, resp. the second of these directly conflicting packages.

Example:

```
package: a
  version: 1
  status: broken
  reasons:
   -
    conflict:
     pkg1:
      package: e
      version: 1
     pkg2:
      package: f
      version: 1
     depchain1:
      -
       depchain:
        -
         package: a
         version: 1
         depends: b
        -
         package: b
         version: 1
         depends: e
     depchain2:
      -
       depchain:
        -
         package: a
         version: 1
         depends: d
        -
         package: d
         version: 1
         depends: f
```

The first part of the dose-debcheck report is as before with details about the broken package. Since this is a conflict, and all conflicts are binary, we give the two packages involved in the conflict first. Packages `f` and `e` are in conflict, but they are not direct dependencies of package `a` . For this reason, we output the

two paths that from a lead to `f` or `e`. All dependency chains for each conflict are together. Again, since there might be more than one way from a to reach the conflicting packages, we can have more then one depchain.

If a conflict occurs between two packages that are both reached through non-trivial dependency chains then we sometimes speak of a *deep conflict*.

4.3 The output in case of co-installability queries

In case of a co-installability query (with the option `--coinst`), the distinction between background and foreground does no longer make sense since the checks now apply to tuples of packages, and not to individual packages. As a consequence, the summary looks a bit different in this case:

Example: In the following example, there are 3 different versions of package `aa`, two different versions of package `bb` and two packages with other names, giving rise to 6 pairs of packages to check for co-installability. Two pairs out of these 6 are found not co-installable:

```
% ./debcheck --coinst "aa,bb" coinst.packages
total-packages: 7
total-tuples: 6
broken-tuples: 2
```

Listings of co-installable, or non co-installable packages when requested with the options `-s`/`--successes`, resp. `-f`/`--failures`, are similar as before but now start on the word `coinst` instead of `package`. Explanations are as before:

Example:

```
% ./debcheck --coinst "aa,bb" -s -f -e coinst.simple
report:
 -
  coinst: aa (= 2) , bb (= 11)
  status: ok
  installationset:
   -
    package: aa
    version: 2
    architecture: all
   -
    package: bb
    version: 11
    architecture: all
   -
    package: cc
    version: 31
    architecture: all
 -
  coinst: aa (= 1) , bb (= 11)
```

```
 status: broken

reasons:
 -
  conflict:
   pkg1:
    package: aa
    version: 1
    architecture: all
    source: aa (= 1)
    unsat-conflict: cc
   pkg2:
    package: cc
    version: 31
    architecture: all
    source: cc (= 31)
   depchain2:

 -
  conflict:
   pkg1:
    package: aa
    version: 1
    architecture: all
    source: aa (= 1)
    unsat-conflict: cc
   pkg2:
    package: cc
    version: 31
    architecture: all
    source: cc (= 31)
   depchain1:

   depchain2:
    -
     depchain:
      -
       package: bb
       version: 11
       architecture: all
       depends: cc


total-packages: 5
total-tuples: 2
broken-tuples: 1
```

5 Exit codes

Exit codes 0-63 indicate a normal termination of the program, codes 64-127 indicate abnormal termination of the program (such as parse errors, I/O errors).

In case of normal program termination:

- exit code 0 indicates that all foreground packages are found installable;
- exit code 1 indicates that at least one foreground package is found uninstallable.

6 Tips and Tricks

6.1 Encoding checks involving several packages

dose-debcheck only tests whether any package in the foreground set is installable. However, sometimes one is interested in knowing whether several packages are co-installable, that is whether there exists an installation set that contains all these packages. One might also be interested in an installation that does *not* contain a certain package.

This can be encoded by creating a pseudo-package that represents the query.

Example: We wish to know whether it is possible to install at the same time `a` and `b`, the latter in some version ≥ 42, but without installing c. We create a pseudo package like this:

```
Package: query
Version: 1
Architecture: all
Depends: a, b(>= 42)
Conflicts: c
```

Then we check for installability of that package with respect to the repository:

```
echo "Package: query\nVersion: 1\nArchitecture: all\nDepends: a, b(>=42)\nConflicts: c"
```

(Beware: This might not do exactly what you want, see below!)

The problem with this encoding is as follows: if we ask dose-debcheck for installability of some package depending on `a` then this dependency can a priori be satisfied by any of the available versions of package `a`, or even by some other package that provides `a` as a virtual package. Virtual packages can be excluded by exploiting the fact that, in Debian, virtual packages are not versioned. As a consequence, any package relation (like Depends) containing a version constraint can only be matched by a real package, and not by a virtual package. This means that the dependency on `b (>= 42)` in the above example already can only be matched by a real package. If we also want to restrict dependency on `a` to real packages only without knowing its possible versions, then we may write `Depends: a (>=0) | a(<0)`.

Example: If we wish to know whether it is possible to install at the same time some version of package `a` and some version of package `b`, under the condition that these are real packages and not virtual packages, then we may construct the following pseudo-package and check its installability:

```
Package: query
Version: 1
Architecture: all
Depends: a(>=0) | a(<0), b(>=0) | b(<0)
```

Note that it is in theory possible, though admittedly quite unlikely, that a package has a version number smaller than 0 (example: $0\sim$).

However, if we have several versions of package `a` and several versions of package `b` then the above pseudo-package is installable if it is possible to install at the same time *some version* of `a` and *some version* of `b`. If we want instead to check co-installability of any combination of versions of package `a` with versions of package `b` then the `--coinst` option (see Section 3.3) is better suited for the task.

6.2 Parsing dose-debcheck's output in Python

Debcheck's output can be easily parsed from a Python program by using the YAML parser (needs the Debian package `python-yaml`).

Example: If you have run debcheck with the option `-f` (and possibly with the `-s` option in addition) you may obtain a report containing one non-installable package (name and version) per line like this:

```
import yaml

doc = yaml.load(file('output-of-distcheck', 'r'))
if doc['report'] is not None:
  for p in doc['report']:
    if p['status'] == 'broken':
      print '%s %s is broken' (p['package'], p['version'])
```

A complete example of a python script that constructs a set of pseudo-packages, runs dose-debcheck on it, and then processes the output is given in the directory `doc/examples/potential-file-overwrites`.

6.3 Usage as a test in a shell script

Exit codes allow for a convenient integration of installation checks as tests in shell scripts.

Example: Suppose that you want to check installability of all `.deb` files in the current directory with respect to the repository `unstable.packages` before uploading your package described in `mypackage.changes`:

```
find . -name "*.deb" -exec dpkg-deb --info '{}' control \; -exec echo ""\; | \
  dose-debcheck --bg unstable.packages && dput mypackage.changes
```

7 Credits

Jérôme Vouillon is the author of the solving engine. He also wrote the first version of the program (called DEBCHECK and RPMCHECK at that time), which was released in November 2005.

The initial development of this tool was supported by the research project *Environment for the development and Distribution of Open Source software (EDOS)*, funded by the European Commission under the IST activities of the 6th Framework Programme. Further development and maintenance of the software, together with new applications building on top of it, was funded by the research project *Managing the Complexity of the Open Source Infrastructure (Mancoosi)*, funded by the European Commission under the IST activities of the 7th Framework Programme, grant agreement 214898.

The work on this software was partly performed at IRILL, the Center for Research and Innovation on Free Software.

8 Further Reading

The dose-debcheck tool, the underlying theory and its application, was described in [**?**].

The paper [**?**] gives an overview of the theory, and explains how dose-debcheck is used for various aspect of quality assurance in Debian.

Checking the relationships between software components is of course also possible and useful for other models of software packages than Debian packages. In fact, the dose-debcheck tool is only one flavor of a more general tool called dose-distcheck which may perform these checks as well for RPM packages and Eclipse plugins, and in the future possibly for even more formats. These formats have many things in common, and the authors of dose-debcheck are convinced that the right architecture for tools dealing with logical aspects of packages is a modular one. Such a modular architecture should be centered around a common universal format for describing the relationships between packages. This architecture is described in [**?**].

9 Copyright and Licence

published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. A special linking exception to the GNU Lesser General Public License applies to the library, see the precise licence information of `dose` for details.

References

[ACTZ11] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. MPM: a modular package manager. In Ivica Crnkovic, Judith A. Stafford, Antonia Bertolino, and Kendra M. L. Cooper, editors, *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering (CBSE 2011)*, pages 179–188, Boulder, CO, USA, June 2011. ACM. [PDF].

[Lan11] Steve Langasek. Multiarch spec, 2011. Available at `https://wiki.ubuntu.com/MultiarchSpec`.

[MBD+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engeineering (ASE)*, pages 199–208, Tokyo, Japan, 2006. IEEE.

[TZ08] Ralf Treinen and Stefano Zacchiroli. Solving package dependencies: from EDOS to Mancoosi. In *Proceedings of DebConf8: 9th annual conference of the Debian project developers*, Mar del Plata, Argentina, August 2008. [PDF].