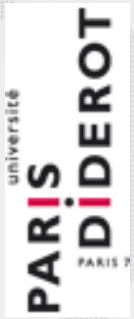
News from EDOS: Finding Outdated Packages

Ralf Treinen

PPS, Université Paris Diderot



Debconf 12, July 14, 2012

Joint work with

Pietro Abate

Roberto Di Cosmo

Zack

Starting point: Edos-debcheck

- Find packages that are not installable
- by looking only at package relations (Depends, Conflicts, ...)
- Use a *complete* solving algorithm (search through all possible alternatives)
- Edos-{dist,deb,rpm}check: fast implementation based on a SAT solver.

Let's run distcheck on the Debian sid

Date	amd64	armel	ia64	i386	mips	mipsel	powerpc	s390	sparc
Fri Sep 2 6:04:12 2011	377 (249)	574 (291)	1321 (238)	394 (269)	847 (204)	902 (271)	530 (298)	510 (229)	515 (264)
Diff with next	+ 46 / - 47	+ 33 / - 25	+ 18 / - 0	+ 66 / - 35	+ 33 / - 61	+ 9 / - 0	+ 45 / - 79	+ 39 / - 26	+ 9 / - 2
Thu Sep 1 6:03:52 2011	378 (268)	566 (293)	1303 (231)	363 (267)	875 (242)	893 (264)	564 (352)	497 (236)	508 (259)
Diff with next	+ 8 / - 55	+ 3 / - 51	+ 2 / - 58	+ 0 / - 55	+ 3 / - 0	+ 66 / - 0	+ 0 / - 0	+ 1 / - 55	+ 0 / - 55
Wed Aug 31 6:03:46 2011	425 (321)	614 (340)	1359 (283)	418 (320)	872 (241)	827 (202)	564 (352)	551 (288)	563 (312)
Diff with next	+ 28 / - 26	+ 25 / - 31	+ 16 / - 34	+ 28 / - 31	+ 23 / - 31	+ 17 / - 35	+ 25 / - 32	+ 23 / - 31	+ 25 / - 31
Tue Aug 30 6:03:39 2011	423 (316)	620 (343)	1377 (301)	421 (320)	880 (245)	845 (216)	571 (355)	559 (292)	569 (314)

Why are there so many not installable packages in sid?

Easy cases

Why are there so many not installable packages in sid?

Easy cases

1. Transient problems that go away when dependencies are built

Why are there so many not installable packages in sid?

Easy cases

1. Transient problems that go away when dependencies are built
2. Packages with Architecture=all that do not have their dependencies satisfied on all architectures.

Why are there so many not installable packages in sid?

Easy cases

1. Transient problems that go away when dependencies are built
2. Packages with Architecture=all that do not have their dependencies satisfied on all architectures.

Not so easy cases

Package p depends on a not installable package, or it depends on packages that conflict, and

3. Not p's fault: the packages that p depends on must be fixed.

Why are there so many not installable packages in sid?

Easy cases

1. Transient problems that go away when dependencies are built
2. Packages with Architecture=all that do not have their dependencies satisfied on all architectures.

Not so easy cases

Package p depends on a not installable package, or it depends on packages that conflict, and

3. Not p's fault: the packages that p depends on must be fixed.
4. p's fault: p has to fix its own dependencies/conflicts in the metadata of a package.

Why are there so many not installable packages in sid?

Easy cases

1. Transient problems that go away when dependencies are built
2. Packages with Architecture=all that do not have their dependencies satisfied on all architectures.

Not so easy cases

Package p depends on a not installable package, or it depends on packages that conflict, and

3. Not p's fault: the packages that p depends on must be fixed.
4. p's fault: p has to fix its own dependencies/conflicts in the metadata of a package.

Goal

Distinguish (3) and (4): Who is to blame when a package is not installable?

How to be sure when it is p's fault?

Idea

When is it the fault of package p in version n that it is not installable in a repository R?

- if (p, n) is not installable in R, and

How to be sure when it is p's fault?

Idea

When is it the fault of package p in version n that it is not installable in a repository R?

- if (p, n) is not installable in R, and
- no matter how all the *other* packages evolve, if package p stays at version n then it will never be installable.

How to be sure when it is p's fault?

Idea

When is it the fault of package p in version n that it is not installable in a repository R?

- if (p, n) is not installable in R, and
- no matter how all the *other* packages evolve, if package p stays at version n then it will never be installable.

Definition

A package (p, n) is *outdated* in a repository R iff (p, n) is not installable in all possible futures of R.

Example 1: Is (*foo*,1) installable?

```
Package: foo
Version: 1
Depends: baz (= 2.5) | bar (= 2.3),
  bar (> 2.6) | baz (< 2.3)

Package: bar
Version: 2

Package: baz
Version: 2
Conflicts: bar (< 3)
```

Example 1: Is (*foo*,1) outdated?

```
Package: foo
Version: 1
Depends: baz (= 2.5) | bar (= 2.3),
  bar (> 2.6) | baz (< 2.3)

Package: bar
Version: 2

Package: baz
Version: 2
Conflicts: bar (< 3)
```

Example 2: Is (*foo*,1) outdated?

```
Package: foo
Version: 1
Depends: baz (= 2.5) | bar (= 2.3),
  bar (> 2.6) | baz (< 2.3)

Package: bar
Version: 2.3

Package: baz
Version: 2.5
Conflicts: bar (> 2.6)
```

What are possible futures of *R*?

Possible Evolutions of a Repository

- Packages may be removed.

What are possible futures of *R*?

Possible Evolutions of a Repository

- Packages may be removed.
- Packages can move to newer versions.

What are possible futures of *R*?

Possible Evolutions of a Repository

- Packages may be removed.
- Packages can move to newer versions.
- Newer versions of packages may change their relations in any way (crude approximation).

What are possible futures of *R*?

Possible Evolutions of a Repository

- Packages may be removed.
- Packages can move to newer versions.
- Newer versions of packages may change their relations in any way (crude approximation).
- New packages may pop up.

What are possible futures of *R*?

Possible Evolutions of a Repository

- Packages may be removed.
- Packages can move to newer versions.
- Newer versions of packages may change their relations in any way (crude approximation).
- New packages may pop up.
- ATM: packages evolve independently of each other.

What are possible futures of *R*?

Possible Evolutions of a Repository

- Packages may be removed.
- Packages can move to newer versions.
- Newer versions of packages may change their relations in any way (crude approximation).
- New packages may pop up.
- ATM: packages evolve independently of each other.

Consequence

There are infinitely many possible futures.

Futures: do we have to care about package removals?

Reasoning

Futures: do we have to care about package removals?

Reasoning

- If (p, n) not installable in any future where we do not have removed packages,
- then (p, n) not installable in any future

Futures: do we have to care about package removals?

Reasoning

- If (p, n) not installable in any future where we do not have removed packages,
- then (p, n) not installable in any future
- Since: Package removal from the *repository* may not make stuff installable.

Futures: do we have to care about package removals?

Reasoning

- If (p, n) not installable in any future where we do not have removed packages,
- then (p, n) not installable in any future
- Since: Package removal from the *repository* may not make stuff installable.

Consequence

We may ignore package removals from R.

Futures: relations of future versions of packages?

Reasoning

Futures: relations of future versions of packages?

Reasoning

- If (p, n) is not installable in any future where new versions of packages have no depends/conflicts,
- then (p, n) is not installable in any future

Futures: relations of future versions of packages?

Reasoning

- If (p, n) is not installable in any future where new versions of packages have no depends/conflicts,
- then (p, n) is not installable in any future
- Since: Adding dependencies and conflicts cannot make stuff installable.

Futures: relations of future versions of packages?

Reasoning

- If (p, n) is not installable in any future where new versions of packages have no depends/conflicts,
- then (p, n) is not installable in any future
- Since: Adding dependencies and conflicts cannot make stuff installable.

Consequence

We may assume that all future versions of packages behave as nicely as possible: no dependencies, no conflicts.

Futures: do we have to care about new packages?

Reasoning

Futures: do we have to care about new packages?

Reasoning

- yes: introducing new packages may make stuff installable,
- but that may happen only if its name is mentioned in a dependency of an existing package.

Futures: do we have to care about new packages?

Reasoning

- yes: introducing new packages may make stuff installable,
- but that may happen only if its name is mentioned in a dependency of an existing package.
- Since: adding packages that noone depends on cannot make stuff installable.

Futures: do we have to care about new packages?

Reasoning

- yes: introducing new packages may make stuff installable,
- but that may happen only if its name is mentioned in a dependency of an existing package.
- Since: adding packages that noone depends on cannot make stuff installable.

Consequence

We only have to consider new packages that are mentioned in dependencies.

What we have so far

When looking at all possible futures . . .

- we have only a finite set of new package names,

What we have so far

When looking at all possible futures . . .

- we have only a finite set of new package names,
- we may ignore package removals,

What we have so far

When looking at all possible futures ...

- we have only a finite set of new package names,
- we may ignore package removals,
- we know what new packages look like (for our purpose): no dependencies, no conflicts

What we have so far

When looking at all possible futures ...

- we have only a finite set of new package names,
- we may ignore package removals,
- we know what new packages look like (for our purpose): no dependencies, no conflicts

Remaining problem

Infinitely many future versions of packages, hence infinitely many future repositories!

How to get finitely many versions

Example

We have package p in version 5.
Other packages have conflicts/dependencies on p :

$$p(\leq 9), p(\neq 12)$$

How to get finitely many versions

Example

We have package p in version 5.
Other packages have conflicts/dependencies on p :

$$p(\leq 9), p(\neq 12)$$

Representative versions

- It is sufficient to consider all the versions that explicitly mentioned:

$$5, 9, 12$$

How to get finitely many versions

Example

We have package p in version 5.
Other packages have conflicts/dependencies on p :

$$p(\leq 9), p(\neq 12)$$

Representative versions

- It is sufficient to consider all the versions that explicitly mentioned:

$$5, 9, 12$$

- plus one between two versions, plus one that is greater than all

$$5, 6, 9, 10, 12, 13$$

Further reduction: observational equivalence

In the example:

- Conflicts/dependencies on p :

$$p(\leq 9), p(\neq 12)$$

- Finitely many versions:

$$5, 6, 9, 10, 12, 13$$

Further reduction: observational equivalence

In the example:

- Conflicts/dependencies on p :

$$p(\leq 9), p(\neq 12)$$

- Finitely many versions:

$$5, 6, 9, 10, 12, 13$$

Observational Equivalence

10 and 13 behave the same, as do 6 and 9:

$$5, 9, 10, 12$$

Are we done, now?

In theory, yes

- We have a finite set (but huge) set F of possible futures.

Are we done, now?

In theory, yes

- We have a finite set (but huge) set F of possible futures.
- With 35.000 packages, two possible versions per package $\Rightarrow 2^{35.000}$ possible futures.

Are we done, now?

In theory, yes

- We have a finite set (but huge) set F of possible futures.
- With 35.000 packages, two possible versions per package $\Rightarrow 2^{35.000}$ possible futures.

Idea

- Put all present and future versions in *one big repository* U.

Are we done, now?

In theory, yes

- We have a finite set (but huge) set F of possible futures.
- With 35.000 packages, two possible versions per package $\Rightarrow 2^{35.000}$ possible futures.

Idea

- Put all present and future versions in *one big repository* U.
- Size: 2×35.000

Are we done, now?

In theory, yes

- We have a finite set (but huge) set F of possible futures.
- With 35.000 packages, two possible versions per package $\Rightarrow 2^{35.000}$ possible futures.

Idea

- Put all present and future versions in *one big repository* U.
- Size: 2×35.000
- U allows precisely the same installations as all the future repositories together

Are we done, now?

In theory, yes

- We have a finite set (but huge) set F of possible futures.
- With 35.000 packages, two possible versions per package $\Rightarrow 2^{35.000}$ possible futures.

Idea

- Put all present and future versions in *one big repository* U.
- Size: 2×35.000
- U allows precisely the same installations as all the future repositories together
- There is one problem with that solution ...

Synchronization

- Binary packages coming from the same source are synchronized !

Synchronization

- Binary packages coming from the same source are synchronized !
- When considering U: we have to exclude installations that mix binary packages coming from the same source but different version.

Synchronization

- Binary packages coming from the same source are synchronized !
- When considering U: we have to exclude installations that mix binary packages coming from the same source but different version.
- Solution: add (versioned!) provides and conflicts:

Synchronization

- Binary packages coming from the same source are synchronized !
- When considering U: we have to exclude installations that mix binary packages coming from the same source but different version.
- Solution: add (versioned!) provides and conflicts:
- If (p, n) has source s: Add
  Provides: src:s $(= n)$
  Conflicts: src:s $(\neq n)$

Synchronization

- Binary packages coming from the same source are synchronized !
- When considering U: we have to exclude installations that mix binary packages coming from the same source but different version.
- Solution: add (versioned!) provides and conflicts:
- If (p, n) has source s: Add
  Provides: src:s $(= n)$
  Conflicts: src:s $(\neq n)$
- We do this only when packages of the same source currently have "similar" version numbers.

Synchronization

- Binary packages coming from the same source are synchronized !
- When considering U: we have to exclude installations that mix binary packages coming from the same source but different version.
- Solution: add (versioned!) provides and conflicts:
- If (p, n) has source s: Add
  Provides: src:s $(= n)$
  Conflicts: src:s $(\neq n)$
- We do this only when packages of the same source currently have "similar" version numbers.
- Finally : One single `distcheck` run on a large repository .

Experiment: sid/main/i386 of 2011/10/06

- 34444 binary packages

Experiment: sid/main/i386 of 2011/10/06

- 34444 binary packages
- Not installable: 431 packages

Experiment: sid/main/i386 of 2011/10/06

- 34444 binary packages
- Not installable: 431 packages
- After adding dummies: 82075 package

Experiment: sid/main/i386 of 2011/10/06

- 34444 binary packages
- Not installable: 431 packages
- After adding dummies: 82075 package
- Runs 1m41s

Experiment: sid/main/i386 of 2011/10/06

- 34444 binary packages
- Not installable: 431 packages
- After adding dummies: 82075 package
- Runs 1m41s
- Reports 119 outdated packages

What packages do we find?

```
package: zhone-illume-glue
version: 0-git20090610-7
source: zhone (= 0-git20090610-7)
reasons:
 -
  missing:
   pkg:
    package: zhone-illume-glue
    version: 0-git20090610-7
    unsat-dependency: python (< 2.7)
```

Ignoring the python transition

Just add to the repository a dummy package

```
Package: python
Version: 2.6-1
```

Example: a very old python dependency

```
package: salome
version: 5.1.3-9
source: salome (= 5.1.3-9)
reasons:
 -
  missing:
   pkg:
    package: salome
    version: 5.1.3-9
    unsat-dependency: python (< 2.6)
```

Example: outdated dependency

```
package: asterisk-chan-capi
version: 1.1.5-1
source: asterisk-chan-capi (= 1.1.5-1)
reasons:
 -
  missing:
   pkg:
    package: asterisk-chan-capi
    version: 1.1.5-1
    unsat-dependency: asterisk (< 1:1.8)
```

Example: needs binNMU

```
package: nitpic
version: 0.1-12
source: nitpic (= 0.1-12)
 -
  missing:
   pkg:
    package: nitpic
    version: 0.1-12
    unsat-dependency: binutils (< 2.21.53.20110923)
```

Example: wrong dependencies

```
package: cyrus-admin-2.2
version: 2.4.12-1
source: cyrus-imapd-2.4 (= 2.4.12-1)
 -
  conflict:
   pkg1:
    package: cyrus-admin-2.4
    version: 2.4.12-1
    source: cyrus-imapd-2.4 (= 2.4.12-1)
    unsat-conflict: cyrus-admin-2.2
   pkg2:
    package: cyrus-admin-2.2
    version: 2.4.12-1
    source: cyrus-imapd-2.4 (= 2.4.12-1)
   depchain1:
       package: cyrus-admin-2.2
       version: 2.4.12-1
       depends: cyrus-admin-2.4
```

EDOS, Mancoosi, Dose

- EDOS European project: Jan 2004 $\longrightarrow$ Jun 2007
- Mancoosi European project: Feb 2008 $\longrightarrow$ May 2011
- New implementation: dose
- This tool: debian package `dose-outdated`
- Also has a much improved debcheck: debian-package `dose-distcheck`

What remains to do

- Better classification of results:
  - Cruft (packages no longer built from source)
  - Packages that just need a recompilation-NMU
  - Packages that are involved in an official transition

What remains to do

- Better classification of results:
  - Cruft (packages no longer built from source)
  - Packages that just need a recompilation-NMU
  - Packages that are involved in an official transition
- Improve the analysis itself:
  - A more precise model how packages may evolve?

What remains to do

- Better classification of results:
  - Cruft (packages no longer built from source)
  - Packages that just need a recompilation-NMU
  - Packages that are involved in an official transition
- Improve the analysis itself:
  - A more precise model how packages may evolve?
- Improve explanations