Research challenges from Free Software distributions

Roberto Di Cosmo
roberto@dicosmo.org

Université Paris Diderot and INRIA

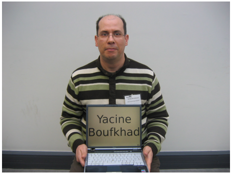
June 8th, 2011
GDR GPL
Lille

Mancoosi at Paris-Diderot

This is joint work with:

Pietro Abate

Jaap Boender



Yacine Boufkhad

Ralf Treinen

Jérôme Vouillon

Stefano Zacchiroli

Outline

1. Context: growing complexity in Open Source
   - Free Software distributions

2. Tools for quality assurance in Free Software Distributions
   - Installation as SAT solving
   - Strong conflicts and antagonistic sets
   - Strong dependencies and dominators
   - Predicting the impact of a package upgrade
   - Predicting the impact of a package upgrade by future version, clustered

Reminder: FOSS

free (as in free *beer*, or *gratuit*) software which has not (yet) to be payed

free (as in *free speech*, or *libre*) software granting 4 freedoms to its users:[1]

0. freedom to *use* the software
1. freedom to *study* and *adapt* the software to user needs (source code)
2. freedom to *distribute* software copies
3. freedom to *distribute modified* software copies

FOSS is *radically* changing the way software is conceived, developed, maintained, deployed, tested, proven, marketed and sold.

[1]there are of course also obligations, which vary according to the license: GPL, BSD, Mozilla, MIT/X, AGPL, ...

Complex Software Systems

Complex software systems are built from *large number* of components, which have to be deployed together; the most challenging are those that *change frequently*.

The free/open source software (FOSS) infrastructure is a complex system *archetype*:

- no central authority / software architect
- quick (*release early, release often*) and distributed development
- strong component interdependency (because of *software reuse*)
- large code bases freely accessible (for developers, students, researchers, . . .)

Component based systems

Components

Proposed 1968 by Douglas McIlroy as a remedy to the "software crisis".

Some Characteristics of Components:

1. Multiple-use
2. Encapsulated
3. A unit of independent deployment and versioning
4. Composable with other components

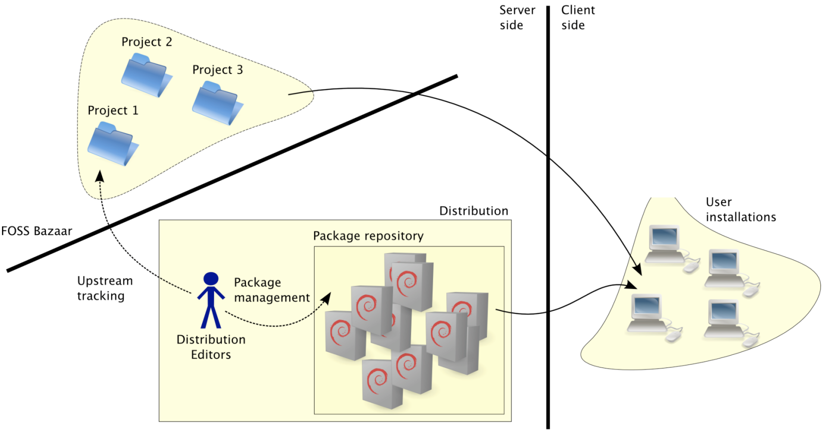
Problem: Interaction between (3) and (4).

GNU/Linux distributions are among the largest component based systems today!

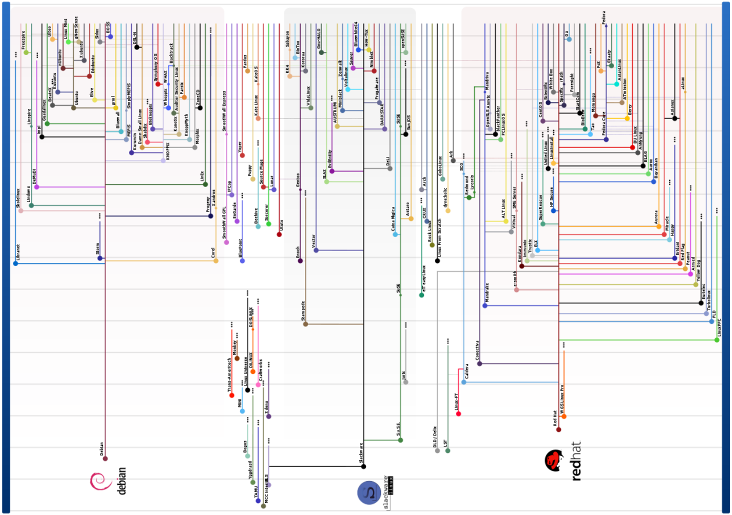
The notion of *distribution*

How do you compose a system by selecting components from dozens of thousands developed independently?

A new idea from FLOSS: GNU/Linux distributions as *intermediaries* between FOSS projects and their users



Distributions: a “somehow” successful idea . . .



Central notion in distributions (to abstract over the complex underlying infrastructure): package, together with package management software . . .

Packages, metadata, installation

Package =
- some files
- some scripts
- metadata
  - Identification
  - Inter-package rel.
    - Dependencies
    - Conflicts
  - Feature declarations
  - Other
    - Package maintainer
    - Textual descriptions
    - ...

Example

```
Package: aterm
Version: 0.4.2-11
Section: x11
Installed-Size: 280
Maintainer: Göran Weinholt ...
Architecture: i386
Depends: libc6 (>= 2.3.2.ds1-4),
  libice6 | xlibs (>> 4.1.0), ...
Conflicts: suidmanager (<< 0.50)
Provides: x-terminal-emulator
...
```

- a package is the *elemental component* of modern distribution systems (not GNU/Linux specific)
- a working *system* is deployed by installing a package set (≈ 1000/2000 for GNU/Linux distro)

Distributions show superlinear growth

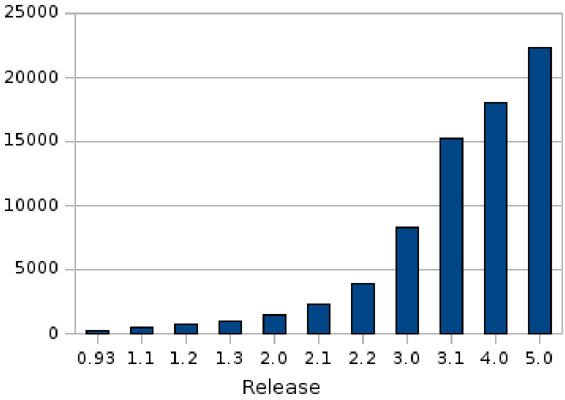


Figure: Number of packages in successive Debian releases

Maintaining and deploying such large collections is becoming hard.

Installation process

Phase	Trace
User request	`# apt-get install aterm`

Constraint resolution	`Reading package lists... Done`
	`Building dependency tree... Done`
	`The following extra packages will be installed:`
	`  libafterimage0`
	`The following NEW packages will be installed`
	`  aterm libafterimage0`
	`0 upgraded, 2 newly installed, 0 to remove and 1786 not upgraded.`
	`Need to get 386kB of archives.`
	`After unpacking 807kB of additional disk space will be used.`
	`Do you want to continue [Y/n]? Y`

Package retrieval	`Get: 1 http://debian.ens-cachan.fr testing/main libafterimage0 2.2.8-2 [301kB]`
	`Get: 2 http://debian.ens-cachan.fr testing/main aterm 1.0.1-4 [84.4kB]`
	`Fetched 386kB in 0s (410kB/s)`

Pre-configuration	

Unpacking	`Selecting previously deselected package libafterimage0.`
	`(Reading database ... 294774 files and directories currently installed.)`
	`Unpacking libafterimage0 (from .../libafterimage0_2.2.8-2_i386.deb) ...`
	`Selecting previously deselected package aterm.`
	`Unpacking aterm (from .../aterm_1.0.1-4_i386.deb) ...`

Configuration	`Setting up libafterimage0 (2.2.8-2) ...` `Setting up aterm (1.0.1-4) ...`

- *each* phase can fail (it actually happens quite often . . .)
- efforts should be made to identify errors as early as possible

Research directions

To improve the situation, there are two main research directions:

solve problems on the *distribution editor's* side

- find broken packages
- find packages which impact large parts of the distribution
- ...

solve problems on the *end user's* side

- optimize the upgrade plan of the user's machine
- design expressive user preference languages
- ...

We focus now on the first part, and leave the rest for a future talk.

The difficult life of distribution maintainers

A distribution maintainer controls the evolution of a distribution by regulating the flow of new packages into it and the removal of packages from it.

With 27000+ packages, we need tools to help, by *efficiently* answering questions like:

1. what are the packages that cannot be installed (*broken*) using the distribution I am releasing?
2. what are the packages that block the installation of many other packages?
3. what are the packages most dependend upon?
4. what are the *broken* packages that can only be fixed by changing them?
5. what are the *future version changes* that will break more packages in the distribution?

Model (simplified, Debian-like)

Names, Versions and Constraints

- Set N of names
- Set V of versions: total and dense order
- Set $\textsc{Con}$ of constraints : $\top$, $= v$, $> v$, $< v$, $\ldots$ where $v \in \mathrm{V}$

A *package* (n, v, D, C) consists of

- a $\mathrm{package\ name}$ n,
- a *version* v,
- a set of dependencies $D \in \mathcal{P}(\mathcal{P}(\mathrm{N} \times \textsc{Con}))$,
- a set of conflicts $C \in \mathcal{P}(\mathrm{N} \times \textsc{Con})$,

A repository

is a set of packages, such that no two different packages carry the same name (*Debian view of the component world*).

An R-installation

is a set $I \subseteq R$ with:

abundance For each element $d \in p.D$ there exists $(n, c) \in d$ and a package $q \in I$ such that $q.n = n$ and $p.v \in [[c]]$.

peace For each $(n, c) \in p.C$ and package $q \in I$, if $q.n = n$ then $q.v \notin [[c]]$.

flatness For all $p, q \in I$: if $p \neq q$ then $p.n \neq q.n$

Installability

$p \in R$ is R-installable if there exists an R-installation I with $p \in I$.

Co-Installability

$S \subseteq R$ is R-co-installable if there exists an R-installation I with $S \subseteq I$.

How hard are the problems related to package installation?

Theorem

The following problems are NP -complete:

- *installability of a single package*
- *coinstallability of a set of packages*

Proof: bi-directional mapping between dependency resolution and boolean satisfiability (see *Di Cosmo, Leroy, Treinen, Vouillon et al, Ase 2006*)

Package installation as a SAT problem

- Version constraints are expanded to the disjunction of the packages in the repository that satisfy that constraint:

$$(p >> 2.1 - 3) \qquad \text{becomes} \qquad (p, 2.2) \vee (p, 2.3 - 1)$$

- For every package P version V in the repository a boolean variable P_v is introduced.
- For every dependency relation we introduce a logical implication of the form $P_v \rightarrow R_1 \wedge \cdots \wedge R_n$
- For every conflict relation we introduce a logical implication of the form $P_v \rightarrow \neg R_1 \wedge \cdots \wedge \neg R_2$
- The encoding of the repository is given by the conjunction of all the logical implication introduced by dependencies and conflicts.
- The flatness of the repository is encoded by explicit conflicts among all the variables of the same name.

Installation and SAT solving

Install `libc6` version `2.3.2.ds1-22` in

```
Package: libc6
Version: 2.2.5-11.8

Package: libc6
Version: 2.3.5-3

Package: libc6
Version: 2.3.2.ds1-22
Depends: libdb1-compat

Package: libdb1-compat
Version: 2.1.3-8
Depends: libc6 (>= 2.3.5-1)

Package: libdb1-compat
Version: 2.1.3-7
Depends: libc6 (>= 2.2.5-13)
```

becomes

$$
\begin{array}{l}
\texttt{libc6}_{2.3.2.ds1-22} \\
\wedge \\
\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.2.5-11.8}) \\
\wedge \\
\neg(\texttt{libc6}_{2.3.2.ds1-22} \wedge \texttt{libc6}_{2.3.5-3}) \\
\wedge \\
\neg(\texttt{libc6}_{2.3.5-3} \wedge \texttt{libc6}_{2.2.5-11.8}) \\
\wedge \\
\neg(\texttt{libdb1-compat}_{2.1.3-7} \wedge \texttt{libdb1-compat}_{2.1.3-8}) \\
\wedge \\
\texttt{libc6}_{2.3.2.ds1-22} \rightarrow \\
(\texttt{libdb1-compat}_{2.1.3-7} \vee \texttt{libdb1-compat}_{2.1.3-8}) \\
\wedge \\
\texttt{libdb1-compat}_{2.1.3-7} \rightarrow \\
(\texttt{libc6}_{2.3.2.ds1-22} \vee \texttt{libc6}_{2.3.5-3}) \\
\wedge \\
\texttt{libdb1-compat}_{2.1.3-8} \rightarrow \texttt{libc6}_{2.3.5-3}
\end{array}
$$

Not that easy: pre-depends, optimizations, error explanation, ...

Special cases

Theorem

If $R = (P, D, C)$ with an empty conflict relation C, then

- *installability of a single package is decidable in linear time*

Proof:

$$P_v \rightarrow ({Q_1}^1 \vee \cdots {Q_1}^{n_1}) \wedge \cdots \wedge ({Q_k}^1 \vee \cdots {Q_k}^{n_k})$$

becomes

$$\begin{array}{ccc} P_v & \rightarrow & ({Q_1}^1 \vee \cdots {Q_1}^{n_1}) \\ & \vdots & \\ P_v & \rightarrow & ({Q_k}^1 \vee \cdots {Q_k}^{n_k}) \end{array}$$

which are dual horn clauses, and the result follows from (the dual of) linear time decidability of satisfiability of Horn formulae (Downing and Gallier, 1984)

Practical complexity

Checking installability is NP-Complete, but recent SAT solvers are able to handle easily current instances.

In Debian, single package installation leads to problems with a few thousands literals, and almost Horn formulae.

The *edos-debcheck* tool, by Jérôme Vouillon, is used daily as part of Debian's Quality Assurance process:

- see it at work on `http://edos.debian.net/weather/` ...
- soon on the Gnome and KDE status bar next to you.

Strong Conflicts

Definition (strong conflicts)

the packages in S are in *strong conflict* if they can never be installed all together

Theorem

Determining whether S is in strong conflict in a repository R is co-NP-complete

Proof: by duality with co-installability.

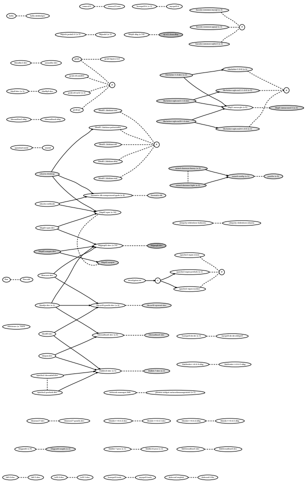
See Boender and Di Cosmo, ISEC 2010

Strong conflicts in Debian Lenny

Strong Conflicts	Package	Explicit Conflicts	Explicit Dependencies	Cone Size	Cone Height
2368	ppmtofb	2	3	6	4
127	libgd2-noxpm	4	6	8	4
127	libgd2-noxpm-dev	2	5	15	5
107	heimdal-dev	2	8	121	10
71	dtc-postfix-courier	2	22	348	8
71	dtc-toaster	0	11	429	9
70	citadel-mta	1	6	123	9
69	citadel-suite	0	5	133	9
66	xmail	4	6	105	8
63	apache2-mpm-event	2	5	122	10
63	apache2-mpm-worker	2	5	122	10
62	harden	0	4	214	9
62	harden-servers	36	2	103	8
57	gpe	0	31	263	10
56	heimdal-servers	10	16	139	9
55	heimdal-servers-x	2	15	142	9
53	libapache2-mod-php5filter	2	16	129	9
52	dtc-cyrus	2	17	345	8
50	kdepimlibs5-dev	1	6	225	9
46	kdebase-runtime-data-common	2	0	1	1

Computing strong conflicts efficiently

By turning Ubuntu main's 7000 packages and 31000 dependencies into



This is *really difficult*: more details in September at Szeged (*Vouillon and Di Cosmo, ESEC/FSE 2011*)

Finding packages that are heavily depended upon

Other *important* packages are those which are needed by many others. Previous work in the literature focused on:

direct dependencies

a package is important if it is mentioned many times in the repository metadata

transitive closure of direct dependencies (*cone* of a package)

a package is important if it appears in the cone of many other packages

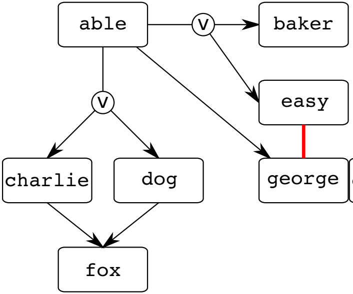
These syntactic notions have no real signification for dependency analysis.

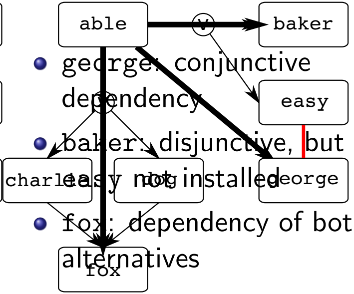
Strong dependencies

Definition

- p strongly depends on q with respect to R if it is not possible to install p without also installing q.
- Impact Set of a package p

$$IS(p) = \{q \in R | q \rightarrow p\}$$





- george: conjunctive dependency
- baker: disjunctive, but easy not installed
- fox: dependency of both alternatives

Strong dependencies

Theorem

Determining whether p strongly depends on q in a repository R is co-NP-complete

Using the boolean formulae encoding of installability, this property can be shown equivalent to proving $p \to q$ in the theory obtained by encoding the repository R.

Strong dependencies are transitive, so the graph of strong dependencies may be huge (and it is: almost one million arrows for Debian Lenny).

Computing Strong dependencies

Naïve approach

```
Sdeps = empty
for p in R
 for q in R
   if check(p) and not sat(p and not q)
   then add (p,q) to Sdeps
 done
done
```

Quadratic number of calls to a sat solver, with $n > 25.000$...
we are not going to do this!

Computing Strong dependencies

We observe that

- if p strongly depends on q, then all installations of p contain q
- a SAT-solver designed for checking installability returns small installations

Smarter algorithm

```
Sdeps = empty
for p in R
 for q in install(p,R)
   if not sat(p and not q) then add (p,q) to Sdeps
 done
done
```

The average size of an installation is small, so the concrete complexity is low, and we can compute strong dependencies for Debian Lenny in a few minutes!

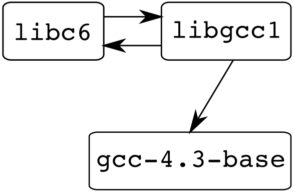
Top 15 of sensitive packages in Debian 5.0

#	**Package**	direct	**strong**	cone
1	`gcc-4.3-base`	43	**20128**	20132
2	`libgcc1`	3011	**20126**	20130
3	`libc6`	10442	**20126**	20130
4	`libstdc++6`	2786	**14964**	15259
5	`libselinux1`	50	**14121**	14634
6	`lzma`	4	**13534**	13990
7	`libattr1`	110	**13489**	14024
8	`libacl1`	113	**13467**	14003
9	`coreutils`	17	**13454**	13991
10	`dpkg`	55	**13450**	13987
11	`perl-base`	299	**13310**	13959
12	`debconf`	1512	**11387**	12083
13	`libncurses5`	572	**11017**	13466
14	`zlib1g`	1640	**10945**	13734
15	`libdb4.6`	103	**9640**	13991

...

GCC

- Why does `gcc-4.3-base` have 43 direct and 20 128 strong predecessors?
- And why does `ligcc1` have 3011 direct and 20 126 strong predessors?
- `libc6`, needed by almost everybody



Strong Dominance

Definition (strong dominance)

We say that p strongly dominates q if :

- p strongly depends on q
- forall o, if o strongly depends on q, then o strongly depends on p

In a picture:

$$\begin{tikzcd}
p_1 \; - - - \arrow[dr] & p_i \; - - - \arrow[d] & p_n \arrow[dl] \\
& q \arrow[d] & \\
& r &
\end{tikzcd}$$

Strong dominance

Theorem

Determining whether p dominates q in a repository R is co-NP-complete

Proof: we can answer the problem with a polynomial algorithm using strong dependencies

Theorem (connection with dominators)

The strong dominators in a repository can be seen as dominators in the detransitivised graph of strong dependencies, augmented with a special start node.

So one can use Tarjan's algorithm on the detransitivised graph.

Approximate strong dominance

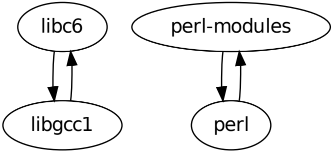
In practice, we use an coarser notion of dominance.

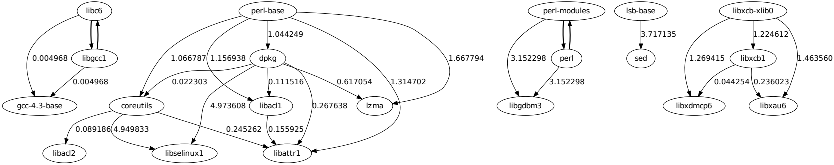
$$\begin{tikzcd}
p_1 \arrow[r, dash, dashed] \arrow[dr] & p_i \arrow[r, dash, dashed] \arrow[d] & p_n \arrow[dl] & s_1 \arrow[r, dash, dashed] \arrow[ddll] & s_k \arrow[ddlll] \\
& q \arrow[d] & & & \\
& r & & &
\end{tikzcd}$$

and put a threshold on the % of nodes on the right w.r.t. the total.
We cannot use Tarjan's algorithm, but the running time is still a few minutes on top of strong dependencies.

Why approximate strong dominance

Using dominators for ordering the top 15 packages from Debian Lenny seen before:





See more details in *Abate, Boender, Di Cosmo and Zacchiroli, ESEM 2009*.

Predicting the future of a distribution

Goal

For a universe U and a package p with version v in it, compute the packages that will be broken if we upgrade p to a future version w, *for all possible future versions* w.

Two main difficulties:

- **changes in** p: we do not know the metadata of any future version w of p... as it is not part of the universe;
- **open ended version space**: the number of possible future version is infinite;

We cannot test all futures! We need to reduce the search space.

Approximating changes in p

Definition (Dummy upgrade package)

$dummy(p, w)$ is a package with name p, version w and with no dependencies and conflicts.

Proposition (Approximation)

Given a universe U containing package p in version v, and a newer version w of package p, then if a package $q \in U$ becomes uninstallable in $upgr(U[(p, v) \mapsto (dummy(p, w))])$, then it is also uninstallable in $upgr(U[(p, v) \mapsto (p, w)])$.

In other words substituting (p, v) with $dummy(p, w)$ is an over approximation of the result of any upgrade of p to any future version w.

Open ended version space

Definition (Constraints of a package)

The list of constraints $constr(p, U)$ of a package p in a universe U is the set of terms $(relop, version)$ associated to p in the conflicts and dependencies constraints of U, taken in lexicographic order.

- **key observation:** Installability only depends on the *valuation* of the constraints (if they are *true* or *false*), and not on the particular version that makes such valuation hold.

Discretization of the future of p

Definition (Valuation of a constraints)

$$eval(c,v) = \begin{cases} v = w & if \quad c \equiv (=, w) \\ v \leq w & if \quad c \equiv (\leq, w) \\ v \geq w & if \quad c \equiv (\geq, w) \\ v < w & if \quad c \equiv (<, w) \\ v > w & if \quad c \equiv (>, w) \end{cases}$$

$$leval([c_1, ..., c_n], v) = [eval(c_1, v)), ..., eval(c_n, v)]$$

Definition (Version equivalence and discriminants)

For a package p in a universe U, consider the ordered *finite* list l of constraints on p in U.

$$v \sim w \iff leval(l, v) = leval(l, w)$$

defines an equivalence relation, with a *finite* set of equivalence classes, on the versions of p. We call *discriminants* of p in U the representatives of

Approximation+Discriminants=Prediction

It is possible to show that

Proposition (Discretisation)

If a universe U contains a package p with version v, and $v \sim w$, then any package q can be installed in U iff it can be installed in $upgr(U[(p,v) \mapsto (p,w)])$.

Corollary (Computing prediction maps)

To approximate the impact of upgrades of a package p, v in U, it is enough to check $upgr(U[(p,v) \mapsto (dummy(p,w))])$ for all discriminants w of p in U.

Algorithm for single package upgrade

$PM \leftarrow []$
for all $p \in U$ **do**
  for all $v_i \in Discriminants(p)$ **do**
    $d \leftarrow dummy(p, v_i)$
    $U' \leftarrow U - \{p\} \cup \{d\}$
    for all $(q, w) \in R$ **do**
      if $\neg check(U', (q, w))$ **then**
        $PM[(p, v_i)] \leftarrow PM[(p, v_i)] \cup \{(q, w)\}$
      end if
    end for
  end for
end for
return PM

Figure: Computing the prediction map of a universe.

This algrithm can run on a full Debian repository in just a few hours.

Results for Lenny

Table: Prediction map for the top 20 Debian impact sets

Package	Version	Target Version	#(IS)	#(BP)
gcc-4.3-base	4.3.2-1.1	$<$ 4.3.2-1.1	20128	20127
libgcc1	1:4.3.2-1.1	$<$ 1:4.3.2-1.1	20126	4
libc6	2.7-18	$<$ 2.7-18	20126	1421
libstdc++6	4.3.2-1.1	any	14964	0
libselinux1	2.0.65-5	$<$ 2.0.65-5	14121	53
lzma	4.43-14	any	13534	0
libattr1	1:2.4.43-2	$<$ 1:2.4.43-2	13489	37
libacl1	2.2.47-2	$<$ 2.2.47-2	13467	36
coreutils	6.10-6	any	13454	0
dpkg	1.14.25	any	13450	0
perl-base	5.10.0-19	$<$ 5.10.0-19	13310	8259
debconf	1.5.24	any	11387	0
libncurses5	5.7+20081213-1	$<$ 5.7+20081213-1	11017	290
zlib1g	1:1.2.3.3.dfsg-12	$<$ 1:1.2.3.3.dfsg-12	10945	582
zlib1g	1:1.2.3.3.dfsg-12	any	10945	0
libdb4.6	4.6.21-11	$<$ 4.6.21-11	9640	12
debianutils	2.30	any	8204	0
libgdbm3	1.8.3-3	$<$ 1.8.3-3	8148	3
sed	4.1.5-6	any	8008	0
perl	5.10.0-19	$<$ 6	7898	775
perl	5.10.0-19	5.10.0-19 $< . <$ 6	7898	775
perl-modules	5.10.0-19	$<$ 5.10.0-19	7898	634

Clustering upgrades

Observation

certain clusters of packages need to be upgraded simultaneously, to avoid breaking too many other packages.

For example, `gcc-4.3-base` is generated automatically from the same source that produces `libgcc1`, and is expected to be upgraded in sync with it. The pointwise analysis misses this fact.

Solution

perform simulated upgrades in clusters

In Debian: cluster along the `Source:` key.
See more details in *Abate and Di Cosmo, HOTSWup 2011*.

Results for Squeeze

Table: Top 20 cluster upgrades, by number of broken components

Source	Version	Target Version	#(BP)
perl	5.10.1-16	$5.10.2 < . < 5.12$	2652
perl	5.10.1-16	$5.10.1\text{-}16 < . < 5.10.2$	2652
perl	5.10.1-16	> 006	2652
perl	5.10.1-16	$5.12 < . < 5.12.0$	2651
perl	5.10.1-16	$5.12.0 < . < 006$	2651
python-defaults	2.6.6-3+squeeze1	> 3	1802
python-defaults	2.6.6-3+squeeze1	$2.07 < . < 2.008$	1800
python-defaults	2.6.6-3+squeeze1	$2.008 < . < 3$	1800
python-numpy	1:1.4.1-5	$> 1:1.5$	542
pygobject	2.21.4+is.2.21.3-1	> 2.21.4+is.2.21.3-1	522
pycairo	1.8.8-1	> 1.8.8-1+b1	517
gtk+2.0	2.20.1-2	> 2.20.1-2	482
udisks	1.0.1+git20100614-3	> 1.1.0	417
eglibc	2.11.2-7	> 2.12	395
eglibc	2.11.2-7	$2.11.2\text{-}7 < . < 2.12$	382
ghc6	6.12.1-13	> 6.12.1+	357
ghc6	6.12.1-13	6.12.1-13 < . < 6.12.1+	357
libnotify	0.5.0-2	> 0.5.0-2	331
ocaml	3.11.2-2	> 3.11.2-2	252
apt	0.8.8	> 0.8.8	219
haskell-mtl	1.1.0.2-10	> 1.1.0.2+	173
haskell-mtl	1.1.0.2-10	1.1.0.2-10+b1 < . < 1.1.0.2+	173
libdbi-perl	1.612-1	> 1.612-1	172
pygtk	2.17.0-4	> 2.17.0-4	129
libjpeg6b	6b1-1	> 6b1-1	115
e2fsprogs	1.41.12-2	> 1.41.12-2	115
mysql-5.1	5.1.49-2	> 5.1.49-2	109
pyorbit	2.24.0-6	> 2.24.0-6	100
[illegible]	[illegible]	[illegible]	[illegible]

Conclusions

Free Software Distributions are a new, interesting area of research:

- full access to real-world problems and information
- research results can be transferred to practice quite quickly (see `edos.debian.net` and `mancoosi.debian.net`)
- there are nice connections with logic, constraints, software engineering, visualization, ...
- ... and the problems can be quite harder than you would expect!

Questions?

Learn more on `http://www.mancoosi.org` and `http://www.irill.org`.

1. Abate, Boender, Di Cosmo and Zacchiroli, *Strong Dependencies between Software Components*, ESEM 2009
2. Boender and Di Cosmo, *Using strong conflicts to detect quality issues in component-based complex systems*, ISEC 2010
3. Abate and Di Cosmo, *Predicting Upgrade Failures Using Dependency Analysis*, HOTSWup 2011
4. Vouillon and Di Cosmo, *On software component co-installability*, ESEC/FSE 2011