Predicting Upgrade Failures Using Dependency Analysis

Pietro Abate [1], Roberto Di Cosmo [2]

Universitè Paris Diderot, PPS
UMR 7126, Paris, France
[1]pietro.abate@pps.jussieu.fr
[2]roberto@dicosmo.org

***Abstract*— Upgrades in component based systems can disrupt other components. Being able to predict the possible consequence of an upgrade just by analysing inter-component dependencies can avoid errors and downtime. In this paper we precisely identify in a repository the components p whose upgrades force a large set of others components to be upgraded. We are also able to discriminate whether all the future versions of p have the same impact, or whether there are different classes of future versions that have different impacts. We perform our analysis on Debian, one of the largest FOSS distributions.**

I. Introduction

Component-based software architectures [1] have the property of being upgradeable pointwise, without necessarily modifying all the components at the same time. Ideally one would like to obtain a coherent software system when replacing any component by any other version of it, but in practice this is rarely possible, and most component systems provide rich metadata to specify under which conditions on the context a component can or cannot be installed.

As a consequence, an upgrade of a single component p from a given version v to a new version w may end up forcing the upgrade of many other components at the same time, and the more pieces are affected by a single upgrade, the higher the impact of the upgrade can be on the usual operations performed by the overall system; this impact can either be beneficial (if works as planned) or disastrous (if not).

We want to identify precisely in a repository the components p whose upgrades force a large set of others components to be upgraded too, and we want also to know whether all the future versions w of p have the same impact, or whether there are different classes of future versions that have different impacts. For the maintainers of component repositories, this information allows to spot the packages and upgrade paths that need to be particularly tested; for the final users, this information may be used to decide which upgrade path to choose, when more than one are possible.

In this paper, we use the notion of *strong dependency* introduced in [2] to build an efficient prediction algorithm that analyses the metadata of a component repository and computes for each component p a sound approximation of the number of other components that will be forced to be upgraded by an upgrade of p. We also show that for any component p in the repository there is only a finite set of future versions which are enough to represent the impact of all its possible upgrades, and our algorithm computes them very efficiently.

Our prediction algorithm can be applied to any component based framework, but in this work we focus on Package-based FOSS (Free and Open Source Software) distributions, which are possibly the largest-scale examples of component-based architectures, whose upgrade effects are experienced daily by million of users world-wide, and for which historical data concerning their evolution is publicly available.

The rest of the paper is structured as follows: Section II recalls the basic notions about FOSS distributions, strong dependencies and impact sets; Section III extends the notion of impact set for upgrades; Section IV gives an efficient algorithm to compute the effective impact sets for upgrades of large software repositories; Section V presents the result of the analysis of large Debian repositories, and a variation of impact sets for upgrades of clusters of packages. Section VI discusses related research.

II. FOSS distributions, strong dependencies and impact sets

In FOSS distributions, software components are managed as *packages* [3]. Packages are described with meta-information, which include complex inter-relationships describing the static requirements to run properly on a target system. Requirements are expressed in terms of other packages, possibly with restrictions on the desired versions. Both positive requirements (*dependencies*) and negative requirements (*conflicts*) are usually allowed.

As the example in Figure 1 shows, inter-package relationships can get quite complex, and there are plenty of more complex examples to be found in distributions like Debian. In particular, the language to express package relationships is not as simple as *flat* lists of component predicates, but rather a structured language whose syntax and semantics is expressed by conjunctive normal form (CNF) formulae [4]. In Figure 1, commas represent logical conjunctions among predicates, whereas bars (“ | ”) represent logical disjunctions. Dependencies can be either versioned or unversioned. In the

Partially supported by the European Community’s 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, “Mancoosi” project. Work performed at the IRILL center for Free Software Research and Innovation in Paris, France.

```
1  Package: python-qt3
2  Priority: optional
3  Section: python
4  Installed-Size: 19324
5  Architecture: i386
6  Version: 3.17.4-1
7  Provides: python2.4-qt3, python2.5-qt3
8  Depends: libc6 (>= 2.7-1), libgcc1 (>= 1:4.2.1),
9    libqt3-mt (>= 3:3.3.7), libstdc++6 (>= 4.2.1),
10   libx11-6, libxext6, python (<< 2.6),
11   python (>= 2.4), python-sip4 (<= 4.8),
12   python-sip4 (>= 4.7)
13 Suggests:
14   libqt3-mt-mysql | libqt3-mt-odbc | libqt3-mt-psql,
15   python-qt3-doc, python-qt3-gl
```

Fig. 1. An excerpt of the inter-package relationships of the `python-qt3` providing Python bindings for Qt.

first case, an unversioned dependency is specified only by a package identifier. In the second case, it is specified by a package identifier and a version constraint of the form `(constr,version)` where `constr` is an operator of $<, \leq, >, \geq, =$. Also, indirections by the mean of so-called *virtual packages* can be used to declare feature names over which other packages can declare relationships; in the example (see line 7: "Provides") the package declares to provide the features called `python2.4-qt3` and `python2.5-qt3`.

Definition 2.1 (Installation): A repository R is a set of packages. An *installation* I is a subset of R that respects the following two properties :

abundance: each package in I has its dependencies satisfied by packages in I;

peace: no package in I conflicts with another package in I.

We say that a package p is installable in a repository R if exists an installation $I \subseteq R$ such that $p \in I$.

Checking installability of a single package in a given repository is an NP-Complete problem, but at the same time that experimental results show how modern SAT-Solvers can handle package installation instances easily [4]. In [2], it has been shown how to efficiently compute *strong dependencies* in a FOSS distribution.

Definition 2.2 (Strong dependency): Given a repository R, we say that a package p in R *strongly depends* on a package q, written $p \Rightarrow q$, if p is installable in R and every installation of R containing p also contains q.

Intuitively, p strongly depends on q with respect to R if it is not possible to install p without also installing q. Strong dependencies were used in [2] to propose the *impact set of p* as a first approximation of the packages that may be affected by an upgrade of a package p.

Definition 2.3 (Impact set of a component): Given a repository R and a package p in R, the *impact set* of p in R is the set $Is((,p)) = \{q \in R \mid q \Rightarrow p\}$.

The impact set of p with respect to R is the set of packages of R that it is not possible to install without also installing p, so changing p *may* have an impact on all its impact set; Table I shows the top 20 packages with biggest impact set, as computed for [2].

TABLE I

PACKAGES FROM DEBIAN 5.0, SORTED BY THE SIZE OF THE STRONG IMPACT SET.

#	Package	—IS(p)—
1	`gcc-4.3-base`	20128
2	`libgcc1`	20126
3	`libc6`	20126
4	`libstdc++6`	14964
5	`libselinux1`	14121
6	`lzma`	13534
7	`libattr1`	13489
8	`libacl1`	13467
9	`coreutils`	13454
10	`dpkg`	13450
11	`perl-base`	13310
12	`debconf`	11387
13	`libncurses5`	11017
14	`zlib1g`	10945
15	`libdb4.6`	9640
16	`debianutils`	8204
17	`libgdbm3`	8148
18	`sed`	8008
19	`perl-modules`	7898
20	`perl`	7898

...

III. IDENTIFYING DISCRIMINANT UPGRADES

Unfortunately, the size of the impact set of a package p alone does not tell us much about what will really happen when p is upgraded to a new version: this will depend on how strict is the dependency relation connecting p to the elements of impact set.

If a package q has an unversioned dependency on a package p, then a change in the version of q will have little effect on the other packages depending on p, as the constraint does not depend on the version of q; the same is true for dependencies of the form $p\ (>, v)$ or $p\ (\geq, v)$ with v a version of p older than the current one. On the contrary, dependencies of the form $p\ (=, v)$, $p\ (\leq, v)$ or $p\ (<, v)$ can be easily broken by an upgrade of v.

As we will see later in table II and III, in Debian Lenny an upgrade of `libstdc++6` will not force the upgrade of any of the 14964 packages in its impact set, while any upgrade of `fontconfig-config` will force a change of all the 4739 packages in its impact set; despite the fact that `libstdc++6` has an impact set almost five times bigger than `fontconfig-config`, the distribution maintainers will need to follow the `fontconfig-config` package transitions even more than the `libstc++6` package transitions, at least as far as dependencies are concerned.

This is a clear motivation for computing, given a universe R and a packages p with version v in it, the packages that will be broken if we upgrade p to a future versions w, for all possible future versions w. There are two main difficulties we need to overcome in order to reach this goal:

- **changes in** p: we do not know what will be the metadata of any future version w of p, precisely because, being a future version, it is not part of the universe;

- **open ended version space**: the number of possible future version being infinite, it is not possible to test them all extensively.

A. Approximating upgrades using dummy packages

For the first issue, we build an upgraded universe by replacing the current version of p with a dummy package with name p and the new version w, but no other dependencies or conflicts. Since the new dummy package will be installable in all contexts, packages in the universe that become broken because of this replacement will also be broken when replacing p with a new, non-dummy p with extra dependencies and conflicts.

Definition 3.1 (Dummy upgrade package): Given a package p with version v, and a target upgrade version w, $dummy(p,w)$ is a new package with the same package name as p, the same provides as p, the new version w and with no dependencies and conflicts.

A dummy package is used to *conservatively* simulate the upgrade of package p to a future version.

Using a notation which is standard for environment manipulation [5], we write $U[(p,v) \mapsto (p,w)]$ for a universe U where a package p with version v is replaced by a package p with version w. If $w > v$ we say that (p,w) is an *upgrade* for package (p,v), if $w < v$ we say that it is a *downgrade*.

Definition 3.2 (Universe Update): Given a package (p,v) in a universe U and a package (p,w), the universe U updated replacing (p,v) with (p,w) is

$$U[(p,v) \mapsto (p,w)] = U - \{(p,v)\} \cup \{(p,w)\}$$

Using dummy packages in computing the broken packages in a future upgrade provides an approximation of what will actually happen in an upgrade.

Proposition 3.3 (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 $U[(p,v) \mapsto (dummy(p,w))]$, then it is also uninstallable in $U[(p,v) \mapsto (p,w)]$, assuming the provides field does not change.

B. Identifying discriminating versions

For the second issue, we notice that the satisfaction of a package dependency only depends on the valuation of the version constraints, which is *true* or *false*, and not on the particular version that makes such valuation hold.

Definition 3.4 (Valuation of a constraints): The function $\texttt{eval}(constr,v)$ is defined as follows

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

This function can be lifted to list of constraints as

$$\texttt{leval}([c_1,...,c_n],v) = [\texttt{eval}(c_1,v)),...,\texttt{eval}(c_n,v)]$$

Definition 3.5 (Constraints of a package): Given a package p in a universe U the list of its constraints $constr(p,U)$s defined as the set of terms of the form $(relop, version)$ associated to package p found in the conflicts and dependencies constraints of U, taken in lexicographic order. $relop$ is the usual relation operator list $<, \leq, >, \geq, =$,

This observation allows us to group the infinite set of future version of a package into a finite set of equivalence classes.

Definition 3.6 (Version equivalence): For a package p in a universe U, consider the *finite* list l of constraints of p. We can define an equivalence relation on versions of p as follows

$$v \sim w = \{\texttt{leval}(l,v) = \texttt{leval}(l,w)\}$$

It is possible to prove the following :

Proposition 3.7 (Discretisation): If a package q can be installed in a universe U that contains a package p with version v, and $v \sim w$, then q can be installed in $U[(p,v) \mapsto (p,w)]$.

As a consequence, when studying the effect of upgrades of a package p in a universe U, we only need to care about the *representatives* of the equivalence classes of v w.r.t. $\sim$, and not all possible future versions of p. We call such representatives the *discriminants* of p in U. For space limitations, we omit here the details of the algorithm that allows to compute such discriminants, but we can mention the fact that in the worst case one gets $2n+1$ discriminants if n is the number of versions of p mentioned in the universe.

C. Prediction maps

Putting together the results of the two previous sections, we can define the predictions that we can obtain from the analysis of a given universe.

Definition 3.8 (Prediction Map): The prediction map *PM* of a universe U associates to all packages $(p,v) \in U$ and all versions $v_i \in discriminants(p)$ the *maximum* set of packages $\{q_1, \cdots, q_n\} \subseteq U$ that are not installable any more when *any possible implementation* of a new package (p,v_i) replaces (p,v).

Intuitively, a prediction map tells us, for each package (p,v) in a universe U, and for each *relevant* future versions w of p, all the packages that will be *surely* broken by an upgrade of p to w, no matter what the precise dependency metadata of (p,w) will be.

We are now in a position to present the algorithm that will compute a prediction map for the upgrades.

IV. Computing predictions

The algorithm in Figure 2 computes the prediction map for a universe U. For each package p with version v, for each versions v_i in the discriminants of p, we compute how many packages q in the impact set of p will be broken if we upgrade package p from version v to v_i.

As we have seen in the previous section, this provides a sound approximation of the packages that will actually be broken by any possible upgrade of p with an actual package p at version v_i.

```
PM ← []
for all (p,v) ∈ U do
  for all v_i ∈ Discriminants(p) do
    for all q ∈ IS(p) do
      U[(p,v) ↦ (dummy(p,v_i))]
      if ¬check(U′,q) then
        PM[(p,v_i)] ← PM[(p,v_i)] ∪ {q}
      end if
    end for
  end for
end for
return PM
```

Fig. 2. Computing the prediction map of a universe.

Notice that if q is installable in U, but breaks in an updated universe $U[(p,v) \mapsto (dummy(p,w))]$, this means that q really needs (p,v) to be installed in U, so $q \in IS(p)$. This is why in the algorithm of Figure 2 we test only the packages of $IS(p)$, the impact set of p, and not all packages in the universe; this allows a significant performance gain, as many packages have a small impact set.

The function $\texttt{check}(U,q)$ uses a SAT solver to return true if the package q is installable in the universe U or false otherwise. The map PM is the prediction map of the universe U. In the worst case, this algorithm requires checking $n^2 * m(v)$ SAT instances, where n is the number of packages in the universe and $2m(v)+1$ is the number of versions mentioned in the universe. To this must be added the algorithmic cost of computing the impact set of each package in the universe, as described in [2].

V. Experimental results

A. Upgrade Predictions in Debian Lenny

We have performed an extensive experimental evaluation of our algorithm on Debian GNU/Linux, that we have chosen as a case study because Debian is the largest FOSS distribution in terms of number of packages (about 22,000 in the latest stable release) and, to the best of our knowledge, the largest component-based system freely available for study.

TABLE II
Prediction map for the top 20 Debian impact sets

Package	Version	Target Version	#(IS)	#(BP)
gcc-4.3-base	4.3.2-1.1	any	20128	20127
libgcc1	1:4.3.2-1.1	any	20126	4
libc6	2.7-18	any	20126	1413
libstdc++6	4.3.2-1.1	any	14964	0
libselinux1	2.0.65-5	any	14121	53
lzma	4.43-14	any	13534	0
libattr1	1:2.4.43-2	any	13489	37
libacl1	2.2.47-2	any	13467	36
coreutils	6.10-6	any	13454	0
dpkg	1.14.25	any	13450	0
perl-base	5.10.0-19	any	13310	7975
debconf	1.5.24	any	11387	0
libncurses5	5.7+20081213-1	any	11017	290
zlib1g	1:1.2.3.3.dfsg-12	any	10945	573
libdb4.6	4.6.21-11	any	9640	12
debianutils	2.30	any	8204	0
libgdbm3	1.8.3-3	any	8148	3
sed	4.1.5-6	any	8008	0
perl	5.10.0-19	≥ 6	7898	14
perl	5.10.0-19	$5.10.0\text{-}19 < . < 6$	7898	13

TABLE III
Top 20 upgrades sorted by number of broken packages

Package	Version	Target Version	#(BP)	#(IS)
gcc-4.3-base	4.3.2-1.1	any	20127	20128
perl-base	5.10.0-19	any	7975	13310
libmagic1	4.26-1	any	5262	5585
fontconfig-config	2.6.0-3	any	4739	4739
python2.5-minimal	2.5.2-15	any	2468	2470
python-minimal	2.5.2-3	any	2368	2369
libc6	2.7-18	any	1413	20126
python	2.5.2-3	≥ 3	1094	2367
python	2.5.2-3	$2.6 < . < 3$	1090	2367
gconf2-common	2.22.0-1	≥ 2.23	899	899
kdelibs-data	4:3.5.10.dfsg.1-0lenny1	$\geq 4{:}3.5.10.dfsg.2$	729	730
libgnomevfs2-common	1:2.22.0-5	$\geq 1{:}2.23$	664	664
libbonobo2-common	2.22.0-1	any	623	623
libgnomecanvas2-common	2.20.1.1-1	≥ 2.21	611	611
libgnome2-common	2.20.1.1-1	≥ 2.21	596	596
zlib1g	1:1.2.3.3.dfsg-12	any	573	10945
libbonoboui2-common	2.22.0-1	≥ 2.23	532	532
libgnomeui-common	2.20.1.1-2	≥ 2.21	529	529
libxau6	1:1.0.3-3	any	502	6795
libxdmcp6	1:1.0.2-3	any	497	6782

In table II we refine the data of the packages in table I: for each package p in the 20 packages with the biggest impact set, we provide the number of broken packages for each future version w in the discriminants of p.

As one can easily see, there is no clear correlation between the size of the impact set and the number of surely broken packages in an upgrade, so it is more pertinent to single out the upgrades with largest number of broken components.

This is done in Figure III, where we present the top 20 package upgrades sorted by the number of actually broken packages. It is remarkable how this new list is different from the previous one: many new packages are brought to our attention, despite their relatively small impact set.

For many packages in this list all future version are equivalent and upgrading to any of them breaks the same number of components, but there are quite a few notable exceptions: for `python`, upgrading to a version strictly greater than `2.06` and strictly smaller than 3 will break 1090 packages, moving to version 3 or later breaks 1094, and upgrading to a version greater than the current `2.5.2-3`, but smaller then `2.06`, breaks only 88 packages (this last entry is not shown in the list as it appears way below). Similarly, the packages related to `bonobo`, `gnome` and `kde` become problematic only further in the future, and there is a version span for which they do not wreak havoc on the system.

B. Clustering related packages

The prediction analysis we have performed up to now allows to identify those packages whose upgrade, *if done in isolation*, may break many installed packages, forcing their upgrade too.

In practice, sometimes certain clusters of packages need to be upgraded simultaneously, to avoid breaking too many other packages. For example, consider again `gcc-4.3-base`: this is a binary package that is generated automatically from the same source that produces `libgcc1`, and is expected to be upgraded in sync with it, as a close examination of their interdependencies shows: the pointwise analysis of the previous section is not able to tell us what will happen when performing a real upgrade, that will install new, aligned versions of both packages.

TABLE IV
TOP 25 CLUSTER UPGRADES, BY NUMBER OF BROKEN COMPONENTS

Source	Version	Target Version	#(BP)
python-defaults	2.5.2-3	≥ 3	1079
python-defaults	2.5.2-3	$2.6 \leq . < 3$	1075
e2fsprogs	1.41.3-1	any	139
ghc6	6.8.2dfsg1-1	$\geq$ 6.8.2+	136
libio-compress-base-perl	2.012-1	$\geq$ 2.012.	80
libcompress-raw-zlib-perl	2.012-1	$\geq$ 2.012.	80
libio-compress-zlib-perl	2.012-1	$\geq$ 2.012.	79
icedove	2.0.0.19-1	$>$ 2.1-0	78
iceweasel	3.0.6-1	$>$ 3.1	70
haskell-mtl	1.1.0.0-2	$\geq$ 1.1.0.0+	48
sip4-qt3	4.7.6-1	$>$ 4.8	47
ghc6	6.8.2dfsg1-1	6.8.2dfsg1+ $\leq . <$ 6.8.2+	36
haskell-parsec	2.1.0.0-2	$\geq$ 2.1.0.0+	29
haskell-parsec	2.1.0.0-2	2.1.0.0-2 $< . <$ 2.1.0.0+	26
sip4-qt3	4.7.6-1	4.8	25
haskell-network	2.1.0.0-2	$\geq$ 2.1.0.0+	22
ghc6	6.8.2dfsg1-1	$<$ 6.8.2dfsg1-1	22
pidgin	2.4.3-4	$\geq$ 3.0	20
ghc6	6.8.2dfsg1-1	6.8.2dfsg1-999 $\leq . <$ 6.8.2dfsg1+	20
pcre3	7.6-2.1	$\geq$ 7.8	17
haskell-regex-base	0.93.1-3	$\geq$ 0.93.1+	16
haskell-regex-posix	0.93.1-1	$\geq$ 0.93.1+	14
haskell-time	1.1.2.0-2	$\geq$ 1.1.2.0+	13
haskell-quickcheck	1.1.0.0-2	$\geq$ 1.1.0.0+	13
haskell-hsql	1.7-2	$\geq$ 1.7+	13

We adapted our algorithm to handle upgrades of a cluster of packages instead of just a package at a time: one needs to use an upgrade universe with dummy packages for all packages belonging to the same cluster, and craft some careful modification of the code to maintain performance of execution, that we do not discuss here for lack of space.

In Debian, a good heuristic for clustering packages turned out to be the `Source` tag in the metadata, that identifies binary packages coming from the same source.

The cluster corresponding to the source package `gcc-4.3`, contains 44 different binary package, among which `libgcc1` and `gcc-4.3-base`, and if we upgrade all these 44 components together, we find out that no other package is broken, so an aligned upgrade of this cluster will not force any other package to upgrade.

We can now present the results of this final, refined analysis, that identifies those packages whose upgrade, even if done together with all their related packages, still wreaks havoc on the whole distribution.

Table IV shows results from the same data set where clusters of packages generated from the same source are migrated together.

For some of the packages, the impact of their upgrade on the rest of the system is reduced to nothing: as noticed above, `gcc-4.3-base` now breaks no package if upgraded with all its related packages at once. Some packages which were pinpointed as problematic are still sources of extreme fragility: `python-defaults` still breaks more than 1000 packages.

But, more important, we can now finally see several packages that will actually be responsible for serious trouble, and that were hidden low in the ranking of the previous measures: one of these is `e2fsprogs` that will break at least 139 packages as soon as we change it for any new version.

It is also interesting to notice that all future versions are not equivalent for most of these packages: for `python-defaults` all upgrades are dangerous, but there is a bump when going to version 3 or higher, and for `ghc6` there will be problems only when upgrading to versions higher than `6.8.2+`.

VI. CONCLUSIONS AND RELATED WORKS

This work follows the research line opened by [2] and [4]. Both works focus on quality assurance of FOSS distributions: In [4] some of the authors give a formalization of the installation problem, various complexity results and an efficient algorithm to check the installability of a single component. In [2] we introduced the notion of strong dependency and impact set as a measure of the relevance of a package with respect to other components in the distribution.

This paper is a significant improvement of our previous results. It provides for the first time a method to estimate faithfully the impact on a distribution of *future* component evolution, be it pointwise, or cluster by cluster, by analysing *present* knowledge of a distribution.

This paper also provides a first connection with the area of quality assurance for large software projects, where many authors correlate component dependencies and past failure rates in order to predict future failures [6], [7], [8]. The underlying

hypothesis is that software "fault-proneness" of a component is correlated to changes in components that are tightly related to it. In particular if a component *A* has many dependencies on a component *B* and the latter changes a lot between versions, one might expect that errors propagates through the network reducing the reliability of *A*. A related interesting statistical model to predict failures over time is the "weighted time damp model" that correlates most recent changes to software fault-proneness [9]. Social network methods [10] were also used to validate and predict the list of *sensitive* components in the Windows platform [6].

Our work still differs for several reasons: the dependency information is not inferred from the source code, but formally declared by the package maintainers, and it is more coarse grained. Second, in FOSS distributions we miss the data needed to correlate upgrade disasters coming from bugs in the components with dependencies and hence to create statistical models that allow to predict future upgrade disasters: this is a future research direction we plan to explore in future work.

VII. Data availability

The data presented in this paper, and much more that was omitted due to space constraints, is available to download from `http://data.mancoosi.org/papers/hostswup2011/`

The tools used to compute the data are released under open source licenses and are available from the Mancoosi website `http://mancoosi.org/software/`. Notice that you do not need our tools to check the results; it is quite straightforward to edit manually a Debian `Packages` file to change the version of a given package, and then run on this modified file the standard `edos-debcheck` tool developed by the EDOS project, and for which a package has been available in Debian for quite a while. A script is available from `http://data.mancoosi.org/papers/hostswup2011/` to check all the data of our experiments presented in this paper.

VIII. Acknowledgements

The authors are very grateful to many people for interesting discussions: all the members of the Mancoosi team at University Paris Diderot, and Julien Cristau and Mehdi Dogguy, from the Debian release team. A special thank goes to the anonymous referees for their remarks, that allowed to improve the paper presentation.

References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 1997.

[2] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli, "Strong dependencies between software components," in *International Symposium on Empirical Sofware Engineering and Measurement*. IEEE Press, Oct. 2009, pp. 89–99. [Online]. Available: http://ieeexplore.ieee.org:80/search/wrapper.jsp?arnumber=5316017

[3] R. Di Cosmo, P. Trezentos, and S. Zacchiroli, "Package upgrades in FOSS distributions: Details and challenges," in *HotSWup'08*, 2008.

[4] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions." in *ASE*, 2006, pp. 199–208.

[5] A. W. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[6] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *ICSE'08*. ACM, 2008, pp. 531–540.

[7] N. Nagappan and T. Ball, "Using software dependencies and churn metrics to predict field failures: An empirical case study," in *ESEM*, 2007, pp. 364–373.

[8] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *ACM Conference on Computer and Communications Security*, 2007, pp. 529–540.

[9] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, 2000.

[10] R. A. Hanneman and M. Riddle, *Introduction to social network methods*, University of California, Riverside, 2005. [Online]. Available: http://www.faculty.ucr.edu/~hanneman/

[11] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998. [Online]. Available: http://dx.doi.org/10.1038/30918