Easing Software Component Repository Evolution *

Jérôme Vouillon
CNRS, PPS UMR 7126,
Univ Paris Diderot,
Sorbonne Paris Cité
jerome.vouillon@pps.univ-paris-diderot.fr

Mehdi Dogguy
EDF S.A.,
Debian Release Team,
Debian Project
mehdi@debian.org

Roberto Di Cosmo
Univ Paris Diderot,
Sorbonne Paris Cité,
PPS, UMR 7126 CNRS, INRIA
roberto@dicosmo.org

ABSTRACT

Modern software systems are built by composing components drawn from large *repositories*, whose size and complexity increase at a fast pace. Maintaining and evolving these software collections is a complex task, and a strict qualification process needs to be enforced to ensure that the modifications accepted into the reference repository do not disrupt its useability. We studied in depth the Debian software repository, one of the largest and most complex existing ones, which uses several separate repositories to stage the acceptance of new components, and we developed `comigrate`, an extremely efficient tool that is able to identify the largest sets of components that can migrate to the reference repository without violating its quality constraints. This tool outperforms significantly existing tools, and provides detailed information that is crucial to understand the reasons why some components cannot migrate. Extensive validation on the Debian distribution has been performed. The core architecture of the tool is quite general, and can be easily adapted to other software repositories.

Keywords

component, repository, software lifecycle, open source

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.9 [**Software Engineering**]: Management—*Lifecycle*

1. INTRODUCTION

Component-based software architectures, maintained in a distributed fashion and evolving at a very quick pace have been popularised by the wide adoption of free and open source software (FOSS). These components are usually made available via a *repository*, which are storage locations from which they can be retrieved. A large variety of repositories are available, ranging from specialised ones for components written in a given programming language, like CPAN, Hackage or PyPI, application-specific ones, like the Eclipse Plugin collection [13], and more general repositories like Maven Central [6] or most GNU/Linux distributions. All these component repositories share the common concern of organising the process of integrating changes: new components are regularly added (Debian grew by more than 8000 packages since the last stable release two years ago), outdated versions are being replaced by more recent ones, and superseded or abandoned components get dropped.

To maintain the quality of a repository, it is necessary to set up a formal process that allows to add, update and remove components in a safe place, where they will be tested and qualified, before moving them to the official repository. For large repositories, this process can only be enacted with the help of automated tools. Existing tools are often unable to cope with all the quality requirements that, depending on the needs of the user community of a repository, may vary from basic unit testing to extensive bug tracking to sophisticated integration tests striving to ensure that components can be combined with each other, a property known as *co-installability* [22]. We have studied in depth the process used to evolve the Debian distribution, which has been in place for more than a decade, managing hundreds of thousands of components, called *packages*, for multiple architectures; since it is open to public inspection, we had access to its formal requirements and we collaborated closely with the Debian Release Team, of which the second author is a member, on solving the problems faced by repository maintainers due to the limitations of the current tools.

The Debian evolution process is organised around three repositories (see Figure 1): *stable*, which contains the latest official release and does not evolve anymore (apart for security and critical updates); *testing*, a constantly evolving repository to which packages are added under stringent qualification conditions, and that will eventually be released as the new *stable*; and *unstable*, a repository in which additions, removals and modifications are allowed under very liberal conditions. A stringent set of requirements, which are formally defined, must be satisfied by packages coming from *unstable* to be accepted in *testing* (also known as *package migration*), and the repository maintainers have responsibility for enforcing them with the help of ad hoc tools. Unfortunately, maintainers are currently all too often con-

*This work was partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, `http://www.irill.org`

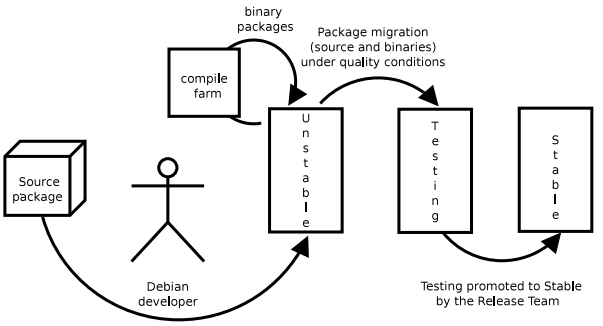


Figure 1: The Debian process (simplified)

fronted with large sets of packages that are stuck in *unstable*, due to complex package interdependencies, with no useful clue to unblock them. A single package can prevent the migration of hundreds of others, and without effective tools to find the culprit, sometimes migration takes months to complete, with a huge amount of manual intervention. In other occasions, the current tool allows into *testing* packages that disrupt co-installability w.r.t. the previous state of the repository, with dire consequences for the users. Indeed, in Debian many applications are split into distinct package that need to be installed together in order to obtain their full functionality, and when this becomes impossible because of an unfortunate modification to the repository, the user need to file bugs, and the maintainers end up spending a significant amount of energy to restore their co-installability.
This article presents `comigrate`, a powerful tool able to efficiently compute maximal sets of packages that can migrate from *unstable* to *testing*: it significantly advances the state of the art by ensuring that no new co-installability issues arise and providing highly valuable explanations for those packages that cannot migrate, helping maintainers find the fix; it is so fast that it can be used *interactively* to narrow down repository issues; it has been validated on various complex migration problems, in collaboration with the Debian Release Team.

The `comigrate` tool is designed using a general architecture, similar to the one of a Sat Modulo Theory solver [19]: it uses a very efficient Boolean solver to quickly identify a large set of packages candidate for migration, starting from general migration criteria that can be encoded as Boolean clauses, and then uses `coinst-upgrade` [23] to look for co-installability issues; when some issues are found, new clauses are added to prevent migration of problematic packages. The process is iterated until a complete solution is found. This general architecture can be reused for building similar tools for other component-based repositories.

The article is structured as follows: in Section 2 we briefly recall basic notions from package based distributions, and give an overview of the Debian integration process with its requirements; Section 3 presents in details an example of how `comigrate` is used to manage a real complex migration; in Section 5, the full architecture of the tool and its algorithm are described; extensions to the algorithm are presented in Section 6; the different ways to use the tools are described in Section 4; Section 7 contains an evaluation of the tool on the Debian integration process; related works are discussed in Section 8 and Section 9 concludes.

2. PACKAGE MIGRATION IN DEBIAN

2.1 Packages and Repositories

Debian software components are called *packages* and come in two flavours: *binary packages* contain the files to be installed on the end user machine, and *source packages* contain all of the necessary files to build these binary packages.

A typical example of the metadata attached to a package is shown in Figure 2, where we can see that the logical language used for expressing dependencies and conflicts is quite powerful, as it allows conjunctions (symbol '`,`'), disjunctions (symbol '`|`') and version constraints (the symbols '`>=`', '`<=`', '`>>`' and '`<<`' stand for the usual $\geq, \leq, >$ and $<$ operators); it is now well known that checking whether a component is installable is an NP-complete problem [2, 4], though real-world instances are tractable [14, 21, 7, 20].

In Figure 2, we find the binary package `ocaml-base` version `3.12.1-4`, which is built from the source package `ocaml` version `3.12.1-4`. Each binary package holds a pointer to its source in the `Source` entry, which may have the same version (in which case only its name is present), or not.

There is a different namespace for source packages and, regarding binary packages, for each architecture, so there can be both a source package and a collection of binary packages named `ocaml`, one for each architecture. In a given namespace, though, there cannot be two packages with the same name and version, and with the notable exception of *unstable*, there can only be one version of a package in any given namespace.

```
1  Package: ocaml-base
2  Source: ocaml
3  Version: 3.12.1-4
4  Architecture: amd64
5  Provides: ocaml-base-3.12.1
6  Depends: ocaml-base-nox, libc6 (>= 2.7),
7    libx11-6, tcl8.5 (>= 8.5.0), tk8.5 (>= 8.5.0),
8    ocaml-base-nox-3.12.1
9
10 Package: ocaml
11 Version: 3.12.1-4
12 Build-Depends: debhelper (>= 8), pkg-config,
13   autotools-dev, binutils-dev, tcl8.5-dev,
14   tk8.5-dev, libncurses5-dev, libgdbm-dev,
15   bzip2, dh-ocaml (>= 1.0.0~)
```

Figure 2: Inter-package relationships of the `ocaml` source component and `ocaml-base`, one of the binary packages generated from it.

The Debian distribution is huge: we gathered the following statistics on February 28, 2013. The number of binary packages is per architecture; an instance of each of these packages is typically built for the thirteen supported architectures. All these instances have to be considered individually during the migration process.

	source packages	binary packages
unstable	18 768	38 903
testing	17 635	36 404
stable	14 968	29 326

A *healthy installation* is a set of packages in which all dependencies are satisfied and with no conflict. If all is well, the set of packages currently installed on your machine under

Debian is a healthy installation. A package is *installable* if there exists at least one healthy installation that contains it. A minimal requirement for a software repository is that all packages are installable. A set of packages are *co-installable* if there exists at least one healthy installation that contains them all. It is normal to have conflicts between some packages, such as for instance between two mail transport agents. Hence, one should not expect all subsets of a software repository to be co-installable. However, when switching to a new version of a software repository, an end user expects to remain able to use all still supported packages. His installation may need to be modified by removing some packages which are no longer supported and by adding new packages, but having to remove any still supported package should remain exceptional. Hence, any set of packages present in both the old and new version of the software repository and which used to be co-installable should normally remain so.

2.2 Package Integration and Migration

The integration process of new packages, and new versions of existing packages, in the Debian distribution involves two repositories, *testing* and *unstable*, and is very complex. We only provide here a general overview and refer the interested reader to [10] for more details.

When a new version of a source package S is available, it is introduced in *unstable*, and then the corresponding binary packages are gradually built and added to *unstable* as well. When a binary package is rebuilt, it replaces the previous version, and when all binary packages are rebuilt, the old version of S is dropped. Binary packages that are no longer built from the new version of S are dropped as well. Building binary packages can be a long process, because of compilation errors and broken dependencies, so it is possible to find in *unstable* several versions of the same source package, and a mixture of binary packages coming from these different versions of the same source.

After a quarantine period, which is useful to detect critical bugs and ranges from zero to ten days according to the priority of a package, the following actions can be performed:

- *replace* a package in *testing* by a package of the same name and a newer version available in *unstable*;
- *remove* a package that no longer exists in *unstable*;
- *add* a new package from *unstable* in *testing*.

Following the Debian Release Team, we call *migration* of a package any of these operations[1].

2.3 Constraints

Package migrations must satisfy many different quality constraints, of different nature. Some of these constraints can be checked by looking only locally at each set of packages built from a source package:

bug reduction a package with new release critical bugs cannot migrate;

binary with source binary and source packages migrate together;

no downgrades packages that migrate have strictly greater version number.

[1]The only anti-intuitive case being the second one, where the migration leads to suppressing a package from *testing*.

Others require a global inspection of the repositories to be assessed:

installability a new package accepted in *testing* should be installable;

no breakage other packages in *testing* should not become non-installable;

co-installability sets of packages that were compatible before migration should remain compatible afterwards.

Often, because of these quality constraints, many packages have to migrate simultaneously: about 490 source packages for the lastest migration of Haskell packages, for instance. When such a large set of packages is stuck, it can be very difficult to manually find out which ones have issues. We need tools to find that out.

2.4 The Current Migration Tool

The tool currently used in Debian to help the Release Team migrate packages from *unstable* to *testing* is a program named `britney`, which heavily relies on heuristics: it first tries to migrate individual source packages together with the associated binary packages, and then looks for clusters of packages that need to migrate together. Since `britney` is unable to find all possible migrations by itself, it provides a complex *hint* mechanism used by the team to help `britney` find sets of packages that can migrate together (the `hint` and `easy` hints) or, when nothing else goes, to force a migration even when this breaks other packages (the `force-hint` hint).

The `britney` tool only checks for installability of individual packages, and not for co-installability of sets of packages, which can have a significant impact on the quality of the repository as we will see in Section 7. It can also be fairly slow and can use up considerable computing resources, but the main complaint from the repository managers is that it provides little help when a migration does not go through.

In the next section we provide a real example of how our replacement tool, `comigrate`, significantly improves over the current state of the art.

3. FINDING MIGRATION ISSUES

On June 13, 2012, we investigated the `ghc` compiler and other associated Haskell packages: the `britney` tool was unable to migrate the `ghc` source package right away, and this prevented the migration of hundreds of related binary pacakges, a most unsatisfactory situation.
The only information at the disposal of the maintainers was the cryptic output from `britney` that looked like:

```
Trying easy from autohinter: ghc/7.4.1-3 ...
leading: ghc,haskell-explicit-exception,haskell-hxt,...
start: 67+0: i-9:a-2:...
orig: 67+0: i-9:a-2:...
easy: 735+0: i-137:a-74:...
    * i386: haskell-platform, haskell-platform-prof, ...
    * amd64: libghc-attoparsec-text-dev, ...
    ...
FAILED
```

with each line containing hundreds of package names. Apart from the very clear last line that stated that the migration failed, there was no hint of what was going wrong.

```
1 > comigrate -c britney2.conf --migrate ghc
2 Package ghc cannot migrate:
3  Package ghc: binary package ghc-haddock/i386 cannot migrate.
4  Package ghc-haddock/i386: needs binary package libghc-happstack-state-doc
5   (would break package libghc-happstack-state-doc):
6   - libghc-happstack-state-doc (testing) depends on haddock-interface-16 {ghc-haddock (testing)}
7  Package libghc-happstack-state-doc/i386: a dependency would not be satisfied
8   (would break package libghc6-happstack-state-doc):
9   - libghc6-happstack-state-doc depends on libghc-happstack-state-doc {libghc-happstack-state-doc (testing)}
```

Figure 3: Output of `comigrate` explaining why package `ghc` cannot migrate.

Our `comigrate` tool is designed to provide valuable help when a migration cannot be performed: running it on exactly the same data, we got the output given in Figure 3, that conveys concisely a wealth of information, using abbreviations and conventions that we detail below, and contains a precious starting hint to explain the situation.

Line 1 says that we are attempting to migrate the source package `ghc`, using the configuration file `britney2.conf`, and line 2 shows that it cannot migrate right away. The rest of the output explains why: line 3 tells us that the source package `ghc` cannot migrate because one of the binary packages built from it, `ghc-haddock`, cannot migrate, at least on architecture `i386`.

In turn, line 4 says that `ghc-haddock` cannot migrate unless the binary package `libghc-happstack-state-doc` migrates (the architecture `i386` is omitted, as it is the same as for `ghc-haddock`). The fact that we read the excerpt `(would break package libghc-happstack-state-doc)` right after, in line 5, means that not migrating the two packages simultaneously would make package `libghc-happstack-state-doc` non installable.

Indeed, the version of `libghc-happstack-state-doc` in *testing* depends on `haddock-interface-16`[2], and we find in line 6, inside the curly braces, all the packages that can satisfy this dependency. As the binary package `ghc-haddock` from *testing* is the only one appearing in the braces, we know that this dependency is not satisfied by the version of `ghc-haddock` in *unstable*, and migrating `ghc-haddock` alone would render `libghc-happstack-state-doc` uninstallable.

But `libghc-happstack-state-doc` cannot migrate: on line 7 and 8 we discover that its migration would break `libghc6-happstack-state-doc`, and looking at the explanation of this fact on line 9, we see that the dependency `libghc-happstack-state-doc` is only satisfied by the version present in *testing*. This tells us that the migration of `libghc-happstack-state-doc` is actually a removal, and that removing it from *testing* breaks an existing package.

Hence, following these few lines of `comigrate` output we learned that migrating package `ghc` is not possible, as it would break `libghc6-happstack-state-doc`.

So we focus on `libghc6-happstack-state-doc` and looking at it we find out that it is in fact a bit special: it contains no file and is just there to ease upgrades when a package is renamed[3]. Following the history of modifications to this package, we discover that at some point in time all the `libghc6-*` packages were renamed into `libghc-*` packages and a single source package `haskell-dummy` was introduced in the distribution to build all the corresponding dummy packages whose role was just to make sure that if somebody needs a `libghc6-*` package, he will actually pull in the corresponding `libghc-*` one. But over time, some `libghc-*` stopped being supported and were removed from *unstable*, while the corresponding `libghc6-*` package were still generated by `haskell-dummy`. It thus seems worthwile to try removing this source package: for simplicity, we focus on the `i386` architecture.

```
> comigrate -c britney2.conf --arches i386 \
      --migrate ghc --remove haskell-dummy
Successful:
age-days 7 haskell-bindings-libzip/0.10-2
# source package xmonad/0.10-4: fix bugs #663470
# source package haskell-cryptocipher/0.3.3-1: fix
#   bugs #674811
age-days 9 haskell-platform/2012.2.0.0
easy [...]
```

This time, the tool has been able to find a way to perform the migration: it is enough to fix two bugs, and to wait for two packages to become old enough.

Making the migration go through on all architectures requires some extra effort, because some binary packages were not built successfully everywhere and have to be removed as well. At the end, we get a list of packages to remove and a large `britney` hint (489 source packages) which make it possible to migrate the `ghc` package. We posted this information on the Debian-release mailing list[4] and it helped successfully migrate the whole set of packages involved.

This concrete example shows how `comigrate` can be used to progressively (depending on the complexity of the migration problem) understand the actions needed to make packages go through.

4. MAIN APPLICATIONS

The `comigrate` tool we have seen at work in Section 3 is highly flexible and very fast (see Section 7 for real world benchmarks). This allows it to be be used for package migration in several ways.

4.1 Automatic Package Migration

By default, `comigrate` computes the largest set of packages that can migrate without breaking any other package or violating any of the constraints defined in the Debian migration process. In this modality, `comigrate` can output the list of packages that should be in *testing* after migration. This allows it to be used as a drop-in replacement for

[2]This is actually a *virtual package*, a named functionalities that can be provided by more than one package, and on which other packages may depend.

[3]This is known as a *transitional dummy package*, see `http://wiki.debian.org/Renaming_a_Package`

[4]`http://lists.debian.org/debian-release/2012/06/msg00317.html`

the `britney` tool for automatic package migration. As an option, `comigrate` can spend extra effort, as explained in Section 6.1, to split the result in clusters of packages that can migrate *independently of each other*. This makes it easier for humans to understand which package can migrate, which packages need to migrate together, and allows maintainers to finely control the process if needed.

Clustered results can also be fed directly into `britney` in the form of `easy` hints, easing the adoption of `comigrate` that can then fruitfully coexist with `britney` for a period.

Sometimes, there are good reasons for some packages to become incompatible after a migration, thus violating the preservation of co-installability. To this end, one can use the `--break` directive, followed by a list of one or more binary packages (for instance, `gnuplot-x11,gnuplot-nox`), to specify that these packages, and *just these packages*, are allowed to become non co-installable. An underscore can be used as a wildcard: package `libjpeg62-dev` was at some point superseded by the incompatible package `libjpeg8-dev`, and one can write `--break libjpeg62-dev,_` to specify that it is allowed to become in conflict with any other package. This option provides a means of controlling which set of packages become non co-installable which is way more precise than other approaches, like the `force-hint` hint of `britney`.

It can also be useful to temporarily remove a package that prevents the migration of other more important packages. In fact, this is a common way to guide package migration. To this end, one can use the `--remove` directive, which corresponds to the homonymous hint of `britney`. Followed by a source package name, it makes the tool behave as if this source package and all its associated binary packages had been removed from *unstable*: `comigrate` will perform the removal of these packages together with the expected migration if this preserves co-installability. An updated version of the removed packages can be put back by a subsequent run of the tool, but in this case, for `comigrate`, these will be new packages, so it will only guarantee that they are installable.

4.2 Explaining Migration Failures

For the packages that cannot migrate, it is very important to provide concise and informative explanations on the reasons that block them. When used with the `--excuses` directive, `comigrate` takes the set of constraints generated by a migration attempt, and outputs an HTML report presenting them in a user-friendly way, giving for each package the precise reasons that prevent its migration. In particular, a graph like the ones shown in Figure 5, in SVG format, is generated for each co-installability issue detected. As we have seen in the real-world example of Section 3, these explanations are precious for finding a way to unblock a migration.

It is important to detect co-installability issues as soon as possible, so that issues regarding a newly introduced package can be immediately brought to the attention of the maintainers, and not only after the quarantine period of ten days. Hence, the migration algorithm is run initially while omitting age and bug constraints, so as to collect as much constraints due to co-installability issues as possible.

4.3 Focusing on a Given Package

As described in Section 3, one can use the `--migrate` directive to focus on the migration of a particular source package. In this modality, `comigrate` drops progressively the less important constraints (age, number of new bugs, out of date packages in *unstable*) that prevents this migration, until either the migration succeeds, or a hard reason to refuse the migration is reached. In case of success, the hints overriding these constraints and the list of packages that need to migrate together are printed. Otherwise, the constraints preventing the migration are printed.

By adding the `--break` and `--remove` directives described above, the user can interact with `comigrate` to get a clear view of all migration issues related to this particular package and find the best possible course of action in order to allow a given source package to migrate.

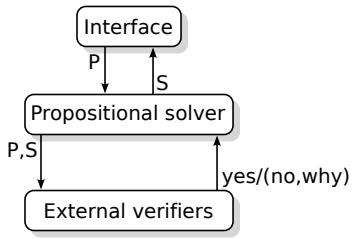


Figure 4: Solver architecture used in `comigrate`

In conclusion, `comigrate` is a sophisticated and flexible tool that allows a full range of modes of operation, from automatic package migration, to fine grained, interactive analysis of the reasons why a package cannot migrate, and progressive relaxation of the migration constraints when needed. Now it's time to look at the internals of `comigrate` and describe how it works.

5. CORE ARCHITECTURE OF THE TOOL

As explained in Section 2.3, the constraints imposed on migration can be broadly separated in two classes. The first one contains constraints that can be easily expressed using Boolean clauses (disjunctive Boolean formulas). Examples of such clauses are "a binary package cannot migrate without its source", or "this binary package can only migrate if these two other binary packages migrate as well." The second class contains constraints that cannot be easily encoded using Boolean clauses, and generally need a global analysis of the repository to be checked: ensuring that a package that migrates to *testing* does not break existing packages requires checking installability for all the packages in the candidate new version of *testing*; ensuring that co-installability is preserved by the migrations needs a special and sophisticated algorithm to be checked efficiently, which is implemented in a separate tool `coinst-upgrade` described in details in [22].

This is a quite interesting situation, that can be handled by using an architecture inspired by the architecture of SMT solvers, and summarised in Figure 4. To find a migration solution for a particular repository configuration P, `comigrate` uses a simple and fast Boolean solver on the constraints that are easily encoded as Boolean clauses and comes up with the largest possible candidate solution S satisfying them all; then, a series of external verifiers are used to check whether other constraints, not easily encoded as Boolean clauses, are satisfied. If this is not the case, these verifiers return an *explanation*, which is used to *learn* some extra Boolean clauses that approximate soundly the constraints, and are added to the original problem, on which the boolean solver is called

again, producing a smaller solution. The process is iterated until a valid solution is found. In our case, there is always at least one solution: performing no migration. Each learnt clause forces at least one additional package not to migrate. Thus, the process eventually terminates.

We now detail each component of this core architecture.

5.1 The Boolean Solver

The encoding of the migration problem uses one Boolean variable per package, with the intended meaning that if the variable is true, the package cannot migrate, and if the variable is false, it can migrate. We have thus to deal with hundreds of thousands of variables (one per package), and the encoding of a typical Debian migration problem is quite big (1 097 490 clauses in the migration on December 18, 2012), so the choice of the Boolean solver has to be done carefully.

We observed that the clauses in the encoding that express constraints like "if some set of packages cannot migrate, then a given package cannot migrate" or assertions like "this package cannot migrate" are actually Horn clauses (Boolean clauses with at most one positive literal). For instance, the encoding of the fact that the binary package `ocaml-base` in Example 2 needs to migrate together with its source `ocaml` is the pair of clauses:

$$\neg\texttt{ocaml} \vee \texttt{ocaml-base}, \qquad \texttt{ocaml} \vee \neg\texttt{ocaml-base}.$$

Indeed, with the exception of the constraints that come from the external verifiers, all clauses in the encoding are Horn clauses, and for this reason, in the current version of the tool, we use a Horn clause solver, that has the following important advantages:

- simplicity: we just need to implement resolution (if the hypotheses of a rule hold, then so does its conclusion); no backtracking is needed;
- speed: it is well known that satisfiability of Horn clauses can be checked in linear time [12];
- optimality: if a set of Horn clauses is satisfiable, then there exists a minimal solution setting to true the least possible number of variables, and which can also be found in linear time [12]; this means that the Horn solver will propose the largest possible migration compatible with the constraints;
- flexibility: it is possible to remove some clauses and update the solver state incrementally to find a new minimal solution;
- easy explanation of why a variable is set: we can explain why a package cannot migrate by printing the tree of Horn clauses which justify it.

In the rare cases when the external verifiers return a Boolean clauses that is not a Horn clause, we approximate it using a stronger Horn clause and when it happens we may not find the optimal solution for the migration. As we will see in Section 7.1, this happens so rarely that it is not an issue in practice, and we have not felt the need to switch to a more sophisticated solver.

In any case, we do not envision any difficulty in replacing this solver with a full blown PMAX-SAT solver or pseudo-Boolean solver, able to handle all kind of Boolean clauses, with the addition of an optimisation function to maximise the number of packages that migrate. The performance should remain good: the fact that we are able to find optimal solutions with the Horn solver means that little backtracking will be needed. But its a significant engineering effort to integrate such a solver with our tool, in particular to still be able to provide readable explanations.

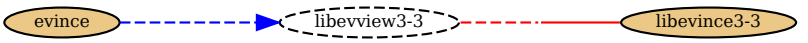
5.2 The External Verifiers

Our Boolean solver specialised for Horn clauses finds a solution that maximises the number of migrating packages given the constraints it knows about. This solution corresponds to a candidate replacement repository for *testing* that lies in-between *testing* and *unstable*, and we need to check whether it contains violations of the second class of quality constraints described in Subsection 2.3: new co-installability issues and new packages that are not installable (old packages that become non installable are an instance of co-installability issues). The verification is performed independently on each of the thirteen architectures supported by Debian, and all the issues found are then used to learn new clauses which are added to the constraints known to the Boolean solver. We detail below the approach taken for each of the two classes of constraints.

New co-installability issues.

The `coinst-upgrades` tool is used to find all co-installability issues introduced in the candidate repository with respect to *testing*. This tool is extensively described in [23], and we just recall here that it takes as input an old and a new repository and computes what is called a *cover* of *broken sets* of packages, which concisely subsumes all co-installability issues introduced in the new repository. A *broken set* is a set of packages which are co-installable in the old repository but no longer in the new one. A *cover* is a collection of *broken sets* such that any healthy installation of packages from the old repository that cannot be successfully upgraded contains at least one of these sets of packages.

All healthy installations can be successfully upgraded if and only if there exists a (unique) empty cover, in which case the check is successful. Otherwise, each broken set is analysed in order to generate a clause to be added to the set of constraints known to the Boolean solver. This analysis was not performed by `coinst-upgrades` and had to be implemented for `comigrate`. As a first step, we extract a small graph of dependencies and conflicts, called *full explanation* in [23], that summarises why the set of packages becomes non co-installable in the candidate repository. Some examples of these explanations, drawn from actual migrations in Debian, are given in Figure 5; they were produced using our tool, with the `--excuses` directive (see Section 4.2 for details). The captions indicate what the tool learns, then how each issue can be fixed by the distribution maintainers. In these graphs, colored packages are the elements of the broken set. Dependencies are represented by arrows, and conflicts by lines connecting two packages. Packages, dependencies and conflicts are drawn with different styles: solid lines indicate an object present in both the old and new repositories, dashed lines indicate an object present only in the new repository, and dotted lines indicate an object present only in the old one. Thus, for instance, on the first graph, `evince` depends in *unstable* on package `libevview3-3` which is only in *unstable*. On the second graph, `mysql-common` is present both in *testing* and *unstable*, but the version in *unstable* conflicts with `mysql-server-core-5.1` both in *testing* and *unstable*.

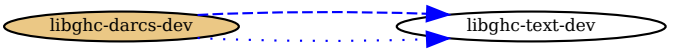


(a) Package `evince` cannot migrate as it depends on a new library which conflicts with the old `libevince3-3` library; this latter library should be removed.

(b) Package `mysql-common` cannot migrate as it conflicts with a previous version of `mysql-server-core`; this old version should be removed.

(c) Package `smarty` cannot be removed as package `piwigo` still depends on it; `piwigo` needs to be removed as well.

(d) Package `libghc-hbro-dev` cannot migrate as it has unsatisfied dependencies; it probably depends on a package that has been updated, and should be recompiled.



(e) Package `libghc-darcs-dev` in *testing* (resp. *unstable*) depends on package `libghc-text-dev` in *testing* (resp. *unstable*), hence both packages must migrate together.

Figure 5: Examples of full explanations.

These small graphs explain why the proposed migration creates a co-installability issue, but do not indicate which parts of the proposed migration are the root cause of the problem. For example, looking at the second graph, we know that the boolean solver has proposed to migrate both `mysql-common` and `mysql-server-core-5.1`, and that this creates an issue. But it would be inefficient to just conclude that they cannot migrate together and learn the clause `mysql-common` $\vee$ `mysql-server-core-5.1`. Indeed, one can see that the actual reason for this problem is the migration of `mysql-common`, as it conflicts with all versions of `mysql-server-core-5.1`, and we can learn the much more informative clause `mysql-common`, which also happens to be a Horn clause, while the first one was not.

To extract a clause from an explanation, we proceeds by iteratively relaxing the constraints on the individual packages in the explanation (allowing some packages to come from either *testing* or *unstable* rather than just from the candidate repository) until we have a minimal set of constraints (packages forced to come from *testing* or *unstable*) that still make the co-installability issue appear. Then, we know that for the co-installability issue to disappear, at least one of the corresponding packages should not take part in the migration. This gives us the Boolean clause we will pass back to the Boolean solver.

If we apply this process on the first four graphs shown in Figure 5, we learn that `evince`, `mysql-common`, `smarty` and `libghc-hbro-dev` should not migrate, which all provide unit clauses. In the last graph, where the proposed migration contains the new version of `libghc-darcs-dev` and the old version of `libghc-text-dev`, the clause is already minimal, and we learn that either `libghc-darcs-dev` should not migrate or `libghc-text-dev` should migrate.

Since the verifier finds co-installability issues caused by the proposed migrations, each Boolean clause contains at least one positive literal, which leads to removing the corresponding package from the migration candidates (a variable set to true means that the corresponding package cannot migrate). If there is a single positive literal, the clause is a Horn clause, and can be passed back to the Boolean solver as is. If there are more than one positive literal (examples are given in Section 7.1), we approximate the clause by only keeping a single positive literal: we know that several packages cannot migrate together and make the decision to keep one back arbitrarily, which may lead to suboptimal solutions.

In order to reduce the risk of making suboptimal choices, we delay any choice resulting in information loss by ignoring non-Horn clauses as long as the analysis of the architecture produces at least a Horn clause.

New non installable packages.

A SAT solver is used to decide which new packages of the candidate repository are not installable. From the output of this solver, we can produce for each non-installable package an explanation of the same shape as the ones produced by `coinst-upgrades`. Then, Boolean clauses can be computed in exactly the same way as described previously.

5.3 Speeding Things Up

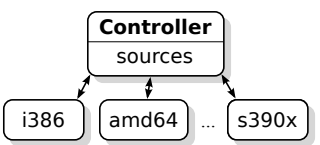


Figure 6: Parallel structure of `comigrate`.

Since the verifiers work independently on 13 different architectures, performance can be greatly improved by using multiple processes as depicted in Figure 6. A main process handles source packages and runs the Horn clause solver. There is one secondary process per architecture. It parses the per-architecture binary package description files and sends to the main process the constraints corresponding to these packages; after that, it behaves as an external verifier for its specific architecture, reading the candidate solution from the main process and sending back clauses to be learnt when issues are found.

6. EXTENSIONS

The algorithm described up to now produces the largest possible migration candidates, except possibly in the rare cases where one non-Horn clause was produced by the external verifiers. It ensures the absence of *co-installability* issues. This core algorithm has been extended to accommodate two additional needs, described below, that lead to interesting variations in the results obtained.

6.1 Clustering Migrating Packages

Instead of writing out the whole set of packages that can migrate as a single huge `easy` hint, an effort is made to cluster packages in sets of packages which can migrate independently of one another, in any order. This approach makes it easier for a human being to understand what a particular migration does; smaller sets are also more resilient to last-minute changes to the repositories while the migration is computed.

Each source package define a group of packages: the source package and its associated binary packages. We need a way to find out which groups need to migrate together to avoid co-installability issues. The idea is to encode in package dependencies all possible configurations of group migrations, and then use an approximation of the `coinst-upgrade` algorithm to find out which configurations may result in co-installability issues. Package dependencies are thus annotated with special literals that indicate whether they hold when a group migrates, or when it does not. For instance the dependency $\texttt{g/old} \vee \texttt{h/new} \vee d$ says that the dependency d must be satisfied when group `g` does not migrate (we have the old version of the group) and group `h` does (we have the new version). We know from [23] that to have a co-installability issue, one must have either a new conflict or a set of new dependencies (which did not exist in *testing*) connected through conflicts. Thus, to avoid co-installability issues, it is enough to put together the groups of packages connected by a new conflict as well as those which occur in pairs of new dependencies connected through a conflict.

This gives us a collection of independent migrations that correspond to the original global migration. We do not claim the result to be minimal (grouping together packages as we do is a sufficient condition, but not a necessary condition to avoid co-installability issues), although in practice the results are quite satisfactory.

As seen in Section 4, `comigrate` can output its results in *hint* format so that it can be used as an external migration solver to help out `britney`. A so-called `easy` hint lists source packages that `britney` should attempt to migrate simultaneously together with their associated binary packages[5].

6.2 Preserving Just Installability

It is possible to run the `comigrate` with an option that makes it preserve just package installability instead of co-installability, even though this is not a good idea in general, as the users of the repository may face serious problems when co-installability is not preserved (see Section 7).

Due to the architecture of the tool, this is just a matter of replacing the external verifier based on `coinst-upgrades`, which computes a collection of sets of packages which are no longer co-installable, with an external verifier that computes a collection of singleton sets of packages which are no longer installable, using for instance the now standard `edos-debcheck` tool [14].

Checking installability may seem a simpler task than checking co-installability, but in fact, surprisingly, this is not the case, and running `comigrate` in such a mode takes longer than in the default mode. Indeed, non co-installability is a property which is more local than installability. For instance, when a conflict is added between two packages `p` and `q`, it is sufficient to report the broken set $\{\texttt{p}, \texttt{q}\}$ when checking co-installability; on the other hand, when looking for new installability issues, one may need to check the installability of the possibly huge number of packages that depend, directly or not, on these two packages to see if any needs both. So, in our case, it turns out that asking for a less precise analysis will actually take more time, even if not prohibilitively more so.

[5]A new version of some binary packages are sometimes built on a given architecture without changing their source package; their migration can be specified by giving a pair formed of the source package name and the architecture. We ignore this case here for the sake of clarity.

7. EVALUATION AND VALIDATION

We have performed a number of experiments and analyses to assess `comigrate`, comparing it with both `britney`, the official tool used in Debian today, and SAT-britney[8], an experimental tool based on a satisfiability solver.

We wanted to check the following points. First, to replace `britney`, our tool should be at least as good at migrating packages; our systematic approach should ensure that our tool can deal with complex migration situations where `britney`'s heuristics fail, but the fact that we do not use a complete Boolean solver could be a concern. Second, a key novelty of our tool is that it can be used interactively to troubleshoot migration issues; it should be fast enough for this task: less than one minute is fine, more than ten minutes is way too slow. Third, our tool is able to perform more stringent checks than `britney` or SAT-britney, preserving not just package installability but also co-installability; we need to verify that this is indeed useful and that the amount of real issues found outweighs the burden of false positives.

To check all this, we have taken a snapshot of all the information needed for package migration twice a day for about two months, from June 24th, 2013 to September 8th, 2013[6] (which gives 152 samples) and we have run the three tools on these configurations. We have also investigated co-installability issues for a longer period, between January 2010 and June 2012 (start of the freeze period), using the historical information on the state of the Debian repositories which is publicly available through the Debian snapshot archive [9]. We could not reproduce accurately migrations over this longer period as some of the required information (number of bugs, package ages, ...) is not archived.

7.1 Tool Comparison: Migrations

We have compared the quality of the migrations performed by `britney` and `comigrate` for the 152 situations mentioned above. To get a meaningful comparison, `comigrate` has been configured with the option to only check for installability. We have not been able to run SAT-britney on all the architectures simultaneously, as its memory usage exceeded the eight gigabytes of RAM available on our testing machines. Hence, we are not able to provide a meaningful comparison with the output of SAT-britney.

We found out that `comigrate` almost always migrates more packages than `britney`, which failed on eleven occasions to find a suitable set of packages that had to migrate together. In fact, we found a single corner case where `comigrate` migrates less packages than `britney`: when the

[6]On June 15th, 2013, after a *freeze* period started in June 2012 during which most migrations were blocked, Debian released its new *stable* repository, and migration were allowed again, so we finally had new, fresh data to analyse.

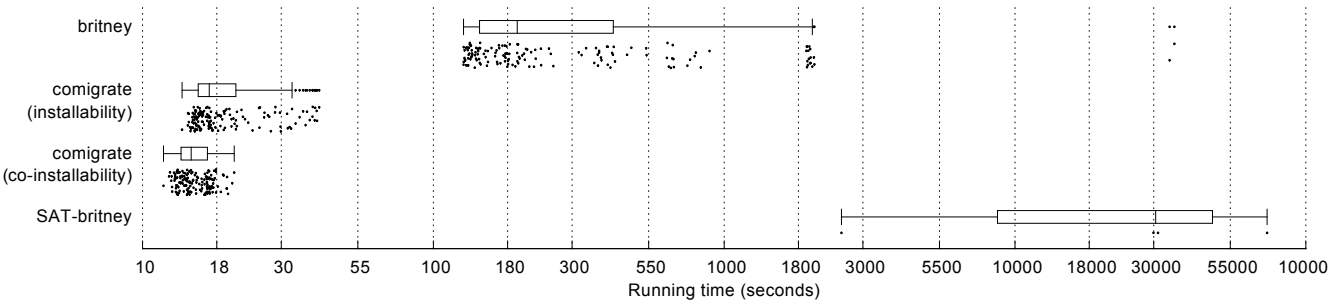


Figure 7: Performance comparison of the different migration tools, on a logarithmic scale. We use standard box plots: the central vertical bar is the median time, the rectangle spans from the first to the third quartiles (it contains half the data), whiskers denote the largest and smallest data within 1.5 times the interquartile range, outliers are represented as isolated dots. Actual data are plotted just below with some vertical jitter.

source used to build a binary package changes, `britney` sometime removes the package from *testing* without waiting for the new version of the package to migrate from *unstable*. In this situation, the extra migration makes the package unavailable for some time, and `comigrate` rightfully refuses to allow it. As a remarkable example, the arrival of KDE 4.10 in *testing* required to migrate 138 packages at once: our tool was able to find automatically which packages had to migrate simultaneously, while a `force-hint` was used by the Debian Release Team to bypass `britney`'s checks and make the packages go through.

We could also verify in all of the 152 runs that `comigrate` never made any suboptimal choice due to the limitations of the solver.

7.2 Tool Comparison: Execution Time

The execution times taken by each tool to compute package migrations, on a 8 GiB desktop computer using an Intel Core i7-870 at 2.93GHz, are shown in Figure 7. The `comigrate` tool takes reliably well less than a minute to compute possible package migrations when only checking for installability issues across upgrades, and is quite faster in the default mode that also checks for co-installability. The `britney` tool is usually one order of magnitude slower, but can occasionally take much longer to complete: we observed a maximum of eight hours over the two month period. We could run SAT-britney on only seven of the thirteen architectures. It is much slower than the other tools, even when running on this limited subset of the distribution, to the point that we aborted the experiment after just a few runs.

7.3 Relevance of Preserving Co-installability

We have compared the output of `comigrate` while checking for co-installability or just installability in the 152 situations mentioned above. We comment shortly on the result. A few packages (`libphonon4`, `liboss4-salsa-asound2` and `libgl1-mesa-swx11`) are incompatible with a huge number of other packages, and there is no point in keeping them co-installable with other packages. A `break` directive can be added once and for all to deal with them. Often, when switching to a new version of a library, the packages of the old version are kept for some time in *testing*. If there is any incompatibility between these packages, `comigrate` will report many harmless conflicts during the transition. This can be avoided by using `break` directives to indicate that the old library is deprecated. We encountered five such transitions. In some case, lots of conflicts are introduced. For instance, on September 10th, there were 16 packages depending on `libopenmpi1.6` and 172 packages depending on the conflicting package `libopenmpi1.3`. It is thus not possible to use any of the 16 packages with any of the other 172 packages. Temporarily introducing incompatibilities this way can sometimes be justified in order to ease package migration, but `britney` does not provide any way to control precisely what incompatibilities are introduced, and member of the Release Team are sometimes not even aware of them. Genuine issues can justify new conflicts between packages. We counted ten of them. For instance, package `quantum` now conflicts with `python-cjson` as the latter is a seriously buggy JSON encoder/decoder; package `parallel` conflicts with `moreutils` as they both provide a binary of the same name but with a different semantics (though, in fact, it is not clear this is allowed at all in Debian); package `lxsession` now replaces package `lxpolkit` and therefore conflicts with it. Sometimes, maintainers can introduce bugs when attempting to fix this kind of issue: the maintainer of `colord` meant to add a conflict with previous versions of `colorhug-client` as the packages contained a common file, which is not allowed, but he added a conflict with the current version as well. In fact, we encountered three instances of overly strong dependencies or conflicts such as the `colord` one. In four cases, the conflicts preventing migration was with a broken or obsolete package that should be removed: `mingw-w64` conflicts with `libpthreads-mingw-w64`, `kde-window-manager` conflicts with `kwin-style-dekorator`. Finally, in six cases, a package migrated when it should have waited for another package remaining in *unstable*. In particular, package `bandwidthd`, meant to be used with a Web server, is in conflict with the version of Apache in *testing*; the FreeBSD kernel `kfreebsd-image-9.1-1-amd64` (`kfreebsd-amd64` architecture) is incompatible with the bootloader `grub-common`; package `scrapbook`, a Firefox extension, is incompatible with Firefox.

We have also checked all co-installability issues introduced each week over two years and a half from January 2010 to June 2012, which we can find thanks to our previous tool `coinst-upgrades` [23], and have assessed their impact. They were most often due to `britney` only checking for installability, rather than being introduced by the Release Team by forcing the migration of a package: we verified this by checking that the culprit packages could migrate without help. We observed that they were real issues in the large majority of

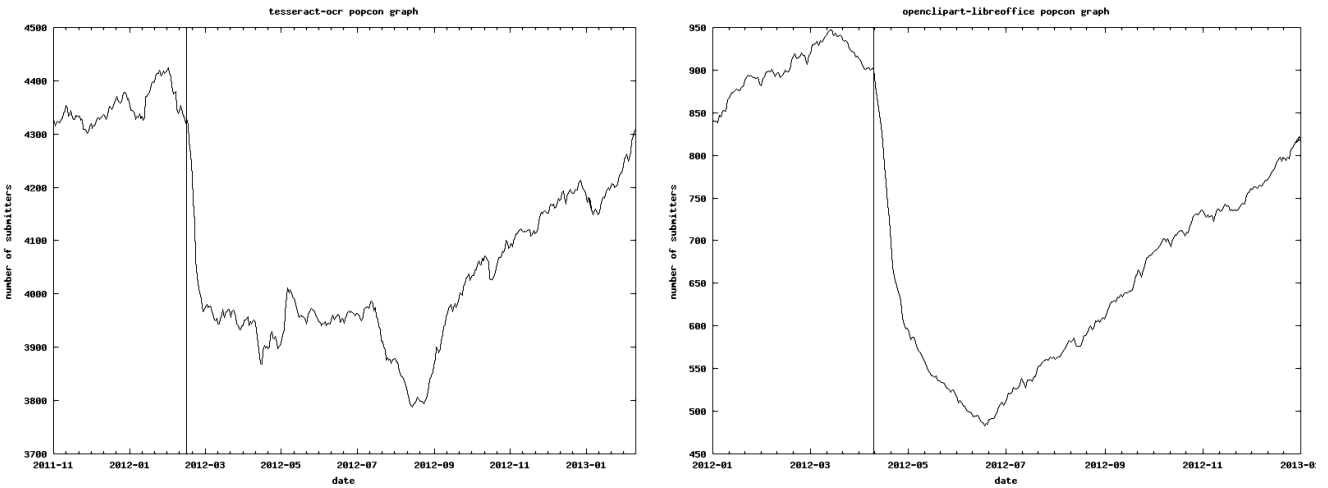


Figure 8: Impact of not preserving co-installability.

cases, where `britney` migrated a package that did not break installability of any other package, and yet made an application unusable. We found about fifteen occurrences of a package migrating in isolation when it should have migrated together with a new version of one or more other packages. In some cases, packages remained kept back in *unstable* for some time, with a disruptive impact on Debian users. We could verify the importance of this impact by analysing the statistics of Debian package installations automatically collected from the users of the `popularity-contest` package, and for which an historical archive is available. The two graphs in Figure 8 show how many users of the package `popularity-contest` have the packages `tesseract-ocr` and `openclipart-libreoffice` installed, over a period of time before and after the disruptive migration takes place. The vertical bar indicates the date of migration of a new version of respectively `tesseract-ocr-eng` (as well as other similar packages) and `openclipart-libreoffice`. These packages conflicts with the versions of their companion packages `tesseract-ocr` and `libreoffice-common` in *testing*, while the new versions of these latter packages remained kept back in *unstable*. One can clearly see that this wrong migration caused `tesseract-ocr` and `openclipart-libreoffice` to be removed on a large fraction of the user base. As a consequence, the application `tesseract` that is split in a common core and a series of language packs became unusable; similarly the Open Clip Art Library could not be used in integration with `libreoffice`.

Overall, checking for co-installability is a significant improvement: while there is some additional work when new conflicts are justified, it does not harm to double check them, and one gains better control on which incompatibilities are introduced, allowing to find a significant number of bugs.

7.4 Validation with the Debian Release Team

A close connection with the Debian Release Team has been established over the past year, and `comigrate` has been shown to consistently provide valid results, and very useful information to track down the reasons why large package sets cannot migrate (the case shown in Section 3 is an actual example). It is now being seriously considered for inclusion in the Debian release management process.

8. RELATED WORK

Ensuring the correct behaviour of components when composed into an assembly is a fundamental concern and has been extensively studied. One may detect automatically behavioural incompatibilities from the component source code [15, 16], or deploy and upgrade such systems [5]. Inter-module dependencies have been used to predict failures [17, 26, 17, 18], identifying *sensitive* components [26] or used as a guideline for testing component-based systems [24, 25].

Maintaining large collections of interrelated software components, and in particular GNU/Linux distributions, poses new challenges, that only recently become subject of research. We know how to identify what other components a package will always need [1] and what pairs of packages are incompatible [11]; sophisticated algorithms allow to answer all the questions about package co-installability [22]. We can identify the component upgrades that are more likely to break installability in a repository [3], and efficiently identify the co-installability issues introduced in a new version of a repository [23]. Package migration is encoded as a PMAX-SAT instance in the `SAT-Britney` tool [8], which only ensures preservation of installability.

9. CONCLUSIONS

We developed `comigrate`, an efficient tool able to compute migration candidates for evolving the Debian repositories according to the stringent quality constraints imposed by the policy, ensuring that no co-installability issues arise, and providing concise and precise explanations when some packages cannot migrate. The unique combination of speed and explanations enable for the first time to perform interactive tuning of the migration process, even when hundreds of interrelated packages are involved. Extensive testing and validation has been performed on real-world problem instances, in collaboration with the Debian release team.

The core architecture of `comigrate` is particularly suitable for handling migrations. It should be straightforward to adapt it to any other integration process where the quality requirements are composed of a mixture of simple constraints and complex conditions verified by external tools.
Source Code is at `http://coinst.irill.org/comigrate`

10. REFERENCES

[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *ESEM*, pages 89–99. IEEE Press, Oct. 2009. doi:10.1109/ESEM.2009.5316017.

[2] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of System and Software Science*, 85(10):2228 – 2240, 2012. Automated Software Evolution. URL: http://www.sciencedirect.com/science/article/pii/S0164121212000477, doi:10.1016/j.jss.2012.02.018.

[3] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Learning from the Future of Component Repositories. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2012)*, Bertinoro, Italie, June 2012. ACM. doi:10.1145/2304736.2304747.

[4] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. A modular package manager architecture. *Information and Software Technology*, 55(2):459 – 474, 2013. Special Section: Component-Based Software Engineering (CBSE), 2011. URL: http://www.sciencedirect.com/science/article/pii/S0950584912001851, doi:10.1016/j.infsof.2012.09.002.

[5] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*, pages 452–476, 2006. doi:10.1007/11785477_26.

[6] Apache Software Foundation. Guide to uploading artifacts to the central repository. [Online; accessed 1-March-2013], 2013. URL: http://maven.apache.org/guides/mini/guide-central-repository-upload.html.

[7] D. L. Berre and A. Parrain. On sat technologies for dependency management and beyond. In *SPLC (2)*, pages 197–200, 2008.

[8] J. Breitner. Tackling the testing migration problem with SAT-solvers, 2012. arXiv:1204.2974.

[9] Debian Project. The debian snapshot archive. [Online; accessed 1-March-2013], 2013. URL: http://snapshot.debian.org/.

[10] Debian Project. Debian "testing" distribution. [Online; accessed 1-March-2013], 2013. URL: http://www.debian.org/devel/testing.

[11] R. Di Cosmo and J. Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC '10: Proceedings of the 3rd India software engineering conference*, pages 163–172, New York, NY, USA, 2010. ACM. doi:10.1145/1730874.1730905.

[12] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Log. Program.*, 1(3):267–284, 1984. doi:10.1016/0743-1066(84)90014-1.

[13] Eclipse Foundation. Eclipse marketplace. [Online; accessed 1-March-2013], 2013. URL: http://marketplace.eclipse.org/.

[14] 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 *Automated Software Engineering (ASE)*, pages 199–208, 2006. doi:10.1109/ASE.2006.49.

[15] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC / SIGSOFT FSE*, pages 287–296, 2003. doi:10.1145/940071.940110.

[16] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, 2004. doi:10.1007/978-3-540-24851-4_20.

[17] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. *ESEM 2007*, pages 364–373, 2007. doi:10.1109/ESEM.2007.13.

[18] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of CCS 2007*, pages 529–540, 2007. doi:10.1145/1315245.1315311.

[19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[20] P. Trezentos, I. Lynce, and A. L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Automated Software Engineering (ASE)*, pages 427–436, 2010.

[21] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *International Conference on Software Engineering (ICSE)*, pages 178–188, 2007.

[22] J. Vouillon and R. Di Cosmo. On software component co-installability. In *SIGSOFT FSE*, pages 256–266, 2011. doi:10.1145/2025113.2025149.

[23] J. Vouillon and R. Di Cosmo. Broken sets in software repository evolution. In *International Conference on Software Engineering (ICSE)*, 2013. URL: http://www.pps.univ-paris-diderot.fr/~vouillon/publi/upgrades.pdf.

[24] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *Automated Software Engineering (ASE)*, pages 409–412, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321696.

[25] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of ISSTA '08*, pages 63–74, New York, NY, USA, 2008. ACM. doi:10.1145/1390630.1390640.

[26] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *International Conference on Software Engineering (ICSE)*, pages 531–540. ACM, 2008. doi:10.1145/1368088.1368161.