Mining Component Repositories for Installability Issues

Pietro Abate*, Roberto Di Cosmo*‡, Louis Gesbert†, Fabrice Le Fessant*†, Ralf Treinen‡, Stefano Zacchiroli‡
*INRIA, Email: pietro.abate@pps.univ-paris-diderot.fr, roberto@dicosmo.org, Fabrice.Le_fessant@inria.fr
†OCamlPro, Email: louis.gesbert@ocamlpro.com
‡Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France,
Email: {ralf,zack}@pps.univ-paris-diderot.fr

***Abstract*—Component repositories play an increasingly relevant role in software life-cycle management, from software distribution to end-user, to deployment and upgrade management. Software components shipped via such repositories are equipped with rich metadata that describe their relationship (e.g., dependencies and conflicts) with other components.**

In this practice paper we show how to use a tool, *distcheck*, that uses component metadata to identify all the components in a repository that cannot be installed (e.g., due to unsatisfiable dependencies), provides detailed information to help developers understanding the cause of the problem, and fix it in the repository.

We report about detailed analyses of several repositories: the Debian distribution, the OPAM package collection, and Drupal modules. In each case, *distcheck* is able to efficiently identify not installable components and provide valuable explanations of the issues. Our experience provides solid ground for generalizing the use of *distcheck* to other component repositories.

I. Introduction

In the last two decades, component repositories have played an important role in many areas, from software distributions to application development. All major Free and Open Source Software (FOSS) distributions are organized around large repositories of software components. Debian, one of the largest coordinated software collections in history [11], contains in its development branch more than 44'000 binary *packages* generated from over 21'000 source packages; the Central Maven repository has a collection of 100'000 Java *libraries*; the Drupal web framework counts over 16'000 *modules*.

Despite different terminologies, and a wide variety of concrete formats, all these repositories use metadata that allows to identify the components, their versions and their interdependencies. Figure 1 shows an example of the metadata of a Debian binary package, highlighting the rich nature of inter-package relationships. In general, packages have both *dependencies*, expressing what must be satisfied in order to allow for installation of the package, and *conflicts* that state which other packages must *not* be installed at the same time. While conflicts are simply given by a list of offending packages, dependencies may be expressed using logical conjunction (written `,`) and disjunctions (`|`). Furthermore, packages mentioned in inter-package relations may be qualified by constraints on package versions.

```
Package: libacl1-dev
Source: acl
Version: 2.2.51-5
Architecture: amd64
Provides: acl-dev
Depends: libc6-dev | libc-dev,
  libacl1 (= 2.2.51-5),
  libattr1-dev (>= 1:2.4.46)
Conflicts: acl (<< 2.0.0), acl-dev,
  kerberos4kth-dev (<< 1.2.2-4)
```

Fig. 1: Example of Debian metadata (excerpt)

The fact that these repositories contain FOSS software, and are widely and freely accessible, make them particularly interesting data sources for mining experiments [19]: various studies have analyzed their growth patterns, according to package size and programming language usage [11], [17], [6], or the evolution of package dependencies and their impact on upgrades [5].

In this article, we focus on a specific quality aspect that concerns large repositories: the amount of components that are not installable, due to the impossibility of satisfying their dependencies and conflicts. These packages can not be used, under any circumstances, so they should either be removed from the repositories, or made installable again by identifying and fixing the reasons of their non-installability. In general, this kind of problems should be detected and fixed *before* it hits the users of the repository.

It turns out that it is possible to perform this kind of analysis by properly exploiting the metadata extracted from component repository using techniques that were first introduced in [16], for the specific case of FOSS distributions like Debian and RedHat. Despite the fact that the underlying problem is NP-complete [3], it was possible to implement several tools which are extremely efficient in practice, and are now used daily in the quality assurance (QA) process for several GNU/Linux distributions [21].

Work partially performed at, and supported by IRILL `http://www.irill.org`. Unless noted otherwise, all URLs in the text have been retrieved on February 13, 2015.

Our contribution

The approach pioneered back then has been generalized and is now embodied in the *distcheck* tool, which can be used to find not installable components across many different kinds of repositories. In this practice article we show how to analyze heterogeneous component repositories for non-installability using *distcheck*, and report our findings on repositories as diverse as: the Debian distribution (Section IV), the OPAM development library collection (Section V), and the software stack of the Drupal web framework (Section VI).

Our findings suggest that, unless a proper quality assurance process is put in place to avoid not installable components, they will remain commonplace in component repositories and affect final users. Thanks to its ability to create reports that pinpoint the causes of non-installability, such a QA process is easy to engineer on top of *distcheck*, as shown by our study of the impact of *distcheck* adoption by the Debian distribution. Hence, we strongly advocate the integration of tools like *distcheck* in the release process of other component repositories.

Code availability

The *distcheck* tool and the other tools used in this paper are implemented in OCaml and part of the Dose library, which is freely released under the terms of the LGPL3 license.

Full information on Dose can be found on our website[1] and precompiled packages are today available in all major GNU/Linux distributions.

II. Theoretical Model

The *distcheck* tool is based on an underlying formal model of inter-component relationships, as they are commonly found in several component models. In this section we briefly recall the main notions of the model. For more details we refer the reader to [16], [2], [3].

A. Components

The metadata associated to a *software component* in most repositories allow to express at least the following features:

- **name**: a long-lived component identifier, stable across software releases, e.g., `libacl1-dev`
- **version**: an identifier for a specific release of a given component, e.g., `2.2.51-5`
- **dependencies**: a description of the additional components that must be installed to make a component usable

The expressiveness of the dependency language varies, but at the very minimum allows for a list of components that are required to be installed. More evolved models also allow for disjunctions (alternatives) and version constraints (like "component c in any version greater than 42").

[1]http://dose.gforge.inria.fr/

Most component models also allow to describe:

- **conflicts**: components that are *not* to be installed at the same time as the given component. Conflicts may come with version constraints, similar to dependencies.
- **features**: names of virtual components *provided by* a component. They may be used to satisfy dependencies of other components and must not conflict with other installed components.

Conflicts are necessary to describe existing incompatibilities among components in a repository: repositories that do not contain this metadata should rely on strict policies to avoid all incompatibilities, or provide tools to install different versions of components in different namespaces [7], otherwise the users will be confronted with serious trouble at installation or execution time.

Features are a convenient way of modularly adding disjunctive dependencies to existing components in a repository.

B. Repositories and installations

A repository R is a set of components, uniquely identified by name and version.

An R-installation I is a set of components, taken from repository R, that enjoys the following two global properties:

- **abundance**: for each component p in I, its dependencies can be satisfied using only components in I;
- **peace**: no components in I conflict with each other.

Definition 1. *We say that a component p is* installable *in a repository R if there exists an R-installation I that contains it.*

Note that component non installability does not just mean that a given component is not installable on top of a set of previously installed components. Such a scenario is not necessarily problematic. Rather, according to the above definition a component is not installable if there is *no subset* of the components shipped by a repository in which the dependencies (and conflicts) of the given component can be respected. *De facto* distributing a non installable component is useless, as no user will ever be able to install it on her machine (without violating its stated inter-component relationships).

The notions of components, repository, and installation can be made formally precise, and have been used to show that checking package installability is an *NP-Complete* problem. Moreover, it is so not only for component models with very rich metadata, like those found in Debian [16], but also for much weaker metadata, like those used in Maven or OSGI [3]. Nevertheless real-world instances of the installability problem turn out to be well-structured, and can be efficiently solved using a variety of approaches [3], ranging from boolean satisfiability solvers to linear integer programming.

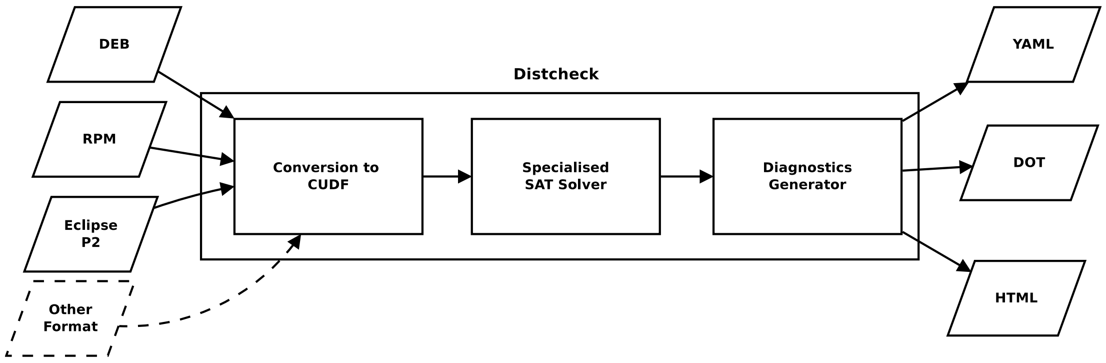


Fig. 2: *distcheck* architecture

III. The Distcheck Tool

The *distcheck* tool is a command line tool, capable of verifying the installability of all (or a selection of) components contained in a given component repository that is given as input.

Internally, *distcheck* is designed as a pipeline, as shown in Figure 2. The front-end on the left is a multiplexer parser that supports several formats for component metadata (Debian `Packages` files, RPM's synthesis or hdlist files, Eclipse OSGI metadata, etc). After metadata parsing, component inter-relationships are internalized in a common data representation called CUDF (Common Upgradability Description Format), an extensible format, with rigorous semantics [22], designed to describe installability scenarios coming from diverse environments without making assumptions on specific component models, version schemas, or dependency formalisms. CUDF can be serialized as a compact plain text format, which makes it easy for humans to read component metadata, and which facilitates interoperability with other component managers that are not yet supported by *distcheck*.

The actual installability check work is performed by a specialized solver, developer by Jérôme Vouillon in 2006, that uses the SAT encoding described in [16] and employs a customized Davis-Putnam SAT solver [9]. Since all computations are performed in-memory and some of the encoding work is shared between all packages, this solver performs significantly faster than a naive approach that would construct a separate SAT encoding for the installability of each package, and then run an off-the-shelf SAT solver on it. For instance, checking installability of all packages of the Debian main repository of the testing suite (for about 40'000 packages) takes about 30 seconds on a commodity 64 bit CPU.

The final component of the pipeline takes the result from the solver and presents it in a variety of human and machine readable formats to the final user. As we will see, an important feature of *distcheck* is its ability, in case a package is found not installable, to produce a concise human-readable explanation that points to the reasons of the issue.

IV. The Debian Distribution

The Debian distribution was started in 1993, it is composed of FOSS software that is packaged by individual developers and teams participating in the Debian project. The metadata used to describe Debian packages is formally defined in [13], and is quite rich: it allows to define dependencies and conflicts, and to specify version constraints on them. Only one version of a given package can be installed in a system at a given moment in time. Recently, the Debian package toolchain has also become *multiarch* aware allowing, for instance, to install i386 packages on an amd64 machine. This recent extension calls for a complex encoding of Debian package metadata into CUDF using the architecture annotations associated to each package.

At each point in time the Debian project offers three branches named "stable", "testing" and "unstable", which are used to implement a rigorous development and quality assurance (QA) process: new packages are usually uploaded to "unstable" first, from which they can migrate to "testing" under several quality conditions. At some point during the release process this migration process is stopped, and "testing" enters a "freeze" phase during which only bug fixes can be applied, until all the quality metrics are met [12] and the "stable" snapshot is taken.

One important issue in Debian quality assurance is the identification and resolution of not installable packages. Since December 2006, *distcheck* is integrated in the Debian quality assurance system, that automatically runs it daily to monitor the state of several Debian distributions (*unstable*, *testing*, *stable*). The results of these daily runs have allowed us to detect and report[2] numerous bugs, most of which have been fixed.

Figure 3 shows the history of not installable packages during the evolution of the Debian distribution, for the "unstable" and "testing" repositories, over the past 9 years. (The "stable" branch is just a snapshot of "testing" taken at the end of the freeze period, and is not shown here: as it gets only security updates, it is not an interesting target for the present study.) We focused on the *i386* architecture, because its history goes

[2]https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=edos-uninstallable;users=treinen@debian.org

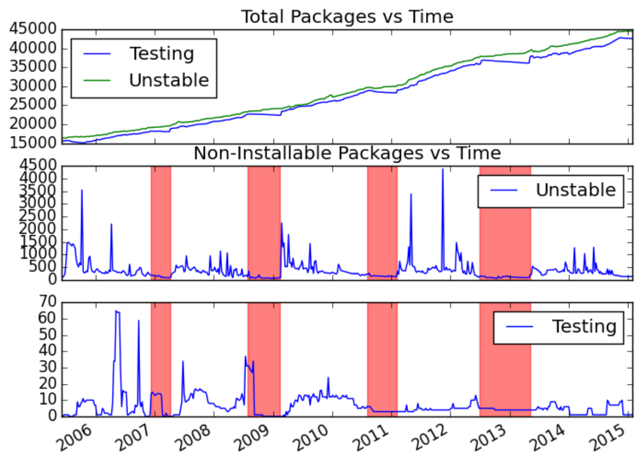


Fig. 3: Time line of not installable packages in Debian

TABLE I: Excerpt of the analysis of the Debian unstable distribution, January 2015

Date	amd64	arm64	armel	armhf	hurd-i386	i386
18 Jan	54/40	28/1008	69/193	52/143	616/2216	60/66
17 Jan	52/39	26/1007	67/192	50/142	616/2216	58/65
16 Jan	52/39	26/1007	67/192	50/142	616/2216	58/65
15 Jan	52/38	26/1006	67/191	50/141	616/2215	58/65
14 Jan	52/38	26/1006	67/191	50/141	616/2215	58/65
13 Jan	51/38	25/1006	66/191	49/141	616/2215	57/65
12 Jan	51/38	25/1006	66/191	49/141	616/2215	57/65

back to 2006; the *amd64* architecture has a shorter history. The topmost graph shows the evolution of the overall number of packages in Debian. The second and third graphs show the number of not installable packages over time. The vertical red regions denote the duration of freezes before a new release. We can observe how the number of not installable packages jumps to relatively high level at the beginning of each release cycle, immediately after the end of a freeze (and hence a release). Then it steadily decreases throughout the release cycle, until reaching almost zero the day of the release.

A live snapshot of our daily analysis for not installable packages is available at the official Debian QA website.[3] Debian builds components repositories for different operating system kernels[4] and CPU architectures, for a total of 13 different OS/CPU pairs (called *architectures* in the following). We run our analysis for all of them, and also aggregate results into cases that are not installable in some of them, or packages that are not installable in any repository where they are available (these are sure bugs).

Some packages are architecture-independent, like documentation packages. These architecture-independent packages are always available in the repositories of all architectures, but may be not installable on architectures where they are not useful[5]. We therefore distinguish these cases. Table I shows an extract from the Debian QA pages. The table shows pairs n/m where n is the number of not installable packages that are specifically built for that architecture, and m is the number of not installable packages which are architecture-independent.

For each of the architectures and aggregations we show the difference between two consecutive runs of our analysis. Packages that are newly found to be not installable are classified into new packages in the archive, and old packages that just have become not installable. Likewise, packages that are no longer found not installable are distinguished between packages that became installable, and packages that have been completely removed from the repository.

It is important to distinguish transient from long-lived installability issues. For instance, given that the *unstable* distribution is just a staging ground for *testing*, temporary failures are normal there. Installability issues in *testing* may also occur, but should be transient.

TABLE II: Not installable packages by duration in Debian *unstable*, February 2015. The arm64 architecture was introduced in August 2014, and hence has no entries yet for 256 days.

Days	amd64	arm64	armel	armf	some	each
16	1/1	1/3	1/1	1/1	1/4	4/0
32	4/4	3/6	4/4	4/4	12/9	1/4
64	1/4	10/58	0/95	0/37	50/118	3/4
128	10/3	11/922	25/13	8/5	504/337	9/5
256	35/24	0/0	36/75	36/91	149/1862	34/15

For this reason we also track the duration for which a package is not installable, and classify the result on a logarithmic time scale (see Table II). Packages which fail to install for an extended period of time on any architecture are sure bugs. We file systematically bug reports against packages that are not installable for at least 64 days on all architectures where they are available. The Debian Bug Tracker allows us to easily follow all these bug reports by using a special *usertag*.

Table III shows a detailed explication of why a package is not installable. In this case, the failure is due to a missing dependency at the end of a dependency chain: `chef-server-webui` in version 10.12.0+dfsg-1 depends on `chef` in at least version 10.12, which is satisfied by the package of that name and version 11.12.8-1. This package in turn depends on `ohai` at least version 6, which is satisfied by

TABLE III: Detailed explanation of why package `chef-server-webui` is not installable on kfreebsd architectures in *unstable*, 2015/02/11.

```
chef-server-webui (10.12.0+dfsg-1)
   ↓chef (>= 10.12)
chef (11.12.8-1)
   ↓ohai (>= 6)
ohai (6.14.0-2)
   ↓ruby-sigar
MISSING
```

[3]http://qa.debian.org/dose
[4]Linux, kFreeBSD, Hurd
[5]e.g., the package `console-setup-freebsd` on Linux architectures

TABLE IV: Detailed explanation of why package `chef-server-webui` is not installable on linux architectures in *unstable*, 2015/02/11.

chef-server-webui (10.12.0+dfsg-1)	
↓ chef (>= 10.12)	↓ chef-server-api (>= 10.12)
	chef-server-api (10.12.0-1)
	↓ chef-solr (>= 10.12.0)
chef (11.12.8-1)	chef-solr (10.12.0+dfsg-2)
CONFLICT	

version 6.14.0-2 of that package. Finally, this package has an unsatisfied dependency on `ruby-sigar`.

Note that explications may become quite complicated, for instance in case a package depends on the *disjunction* of two packages, each of which has an unsatisfied dependency.

For the same package we obtain a different explanation on most of the linux architectures, as shown in Table IV. Here, `chef-server-webui` indirectly depends, via the shown dependency chains, both on `chef` and `chef-solr`. However, these two packages are in conflict.

V. The OPAM Repository

OPAM is a recent package manager for OCaml libraries and tools which is widely used since 2012. OCaml is stricter than other languages on the compatibility of interfaces between binary packages, so OPAM was designed to manage source packages that have to be compiled before installation.

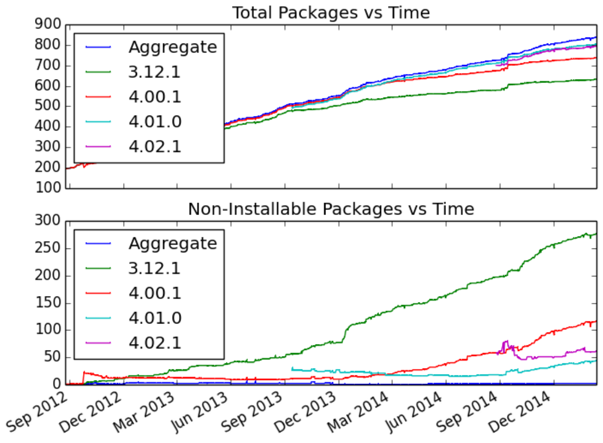
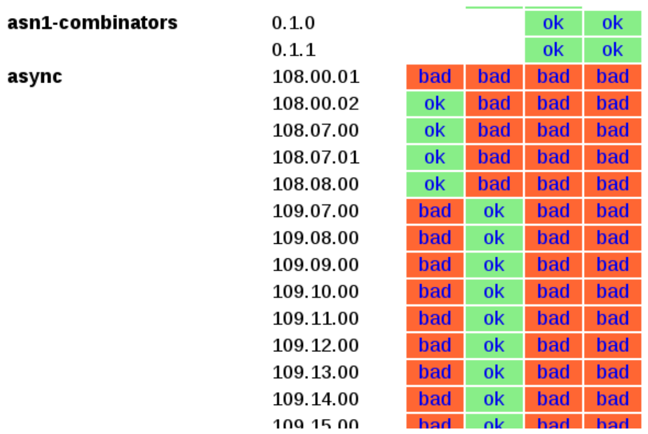


Fig. 4: We plot the evolution of the number of packages over time since OPAM creation, and the evolution of non-installable packages. Despite manual review of updates and automatic testing, non-installable packages keep increasing, showing the need for better Q&A tooling on the repository.

OPAM allows for multiple versions of the OCaml compiler to be installed as different installation trees, called *switches*. In a given switch, only one version of each package may be installed at a given time, but the number of switches is not limited. Internally, OPAM is implemented over the Dose library, and thus uses CUDF to communicate with external solvers. It actually includes Distcheck internally, to provide better error messages on installation conflicts.

The OPAM repository itself is hosted on Github (http://opam.ocaml.org/). On Feb 2015, it contains 836 different packages, and more than 3000 package versions. Updates are received through Github's pull-request mechanism. They are reviewed by a few repository maintainers, after an automatic testing system has shown that updated packages can be installed for every OCaml version they are available for.



asn1-combinators	0.1.0			ok	ok
	0.1.1			ok	ok
async	108.00.01	bad	bad	bad	bad
	108.00.02	ok	bad	bad	bad
	108.07.00	ok	bad	bad	bad
	108.07.01	ok	bad	bad	bad
	108.08.00	ok	bad	bad	bad
	109.07.00	bad	ok	bad	bad
	109.08.00	bad	ok	bad	bad
	109.09.00	bad	ok	bad	bad
	109.10.00	bad	ok	bad	bad
	109.11.00	bad	ok	bad	bad
	109.12.00	bad	ok	bad	bad
	109.13.00	bad	ok	bad	bad
	109.14.00	bad	ok	bad	bad
	109.15.00	bad	ok	bad	bad

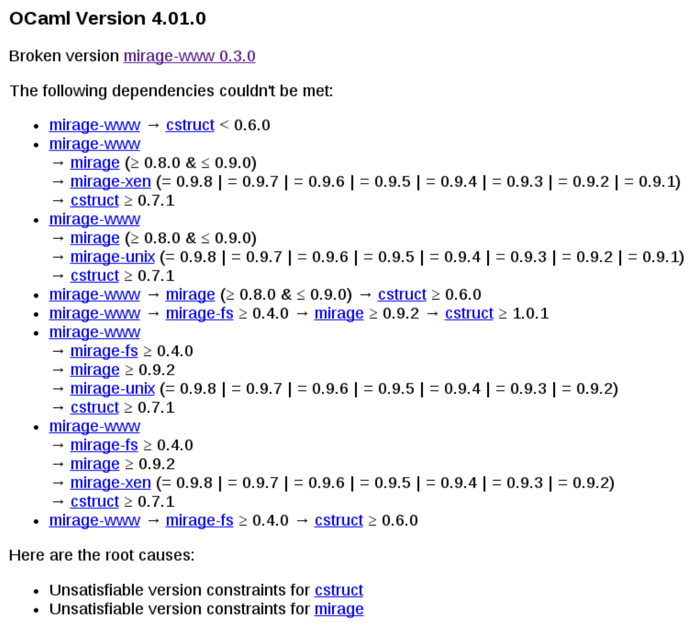
Fig. 5: An excerpt of `http://ows.irill.org/` showing a sample of the summary table. Boxes are white when explicitely disabled, red when otherwise non-installable, and green when installable. Each column is for a different version of OCaml (*switch)*.

Figure 4 plots the evolution of the number of OPAM packages over time. It shows how the tool became popular, growing from 200 packages to 800 packages in two years, but also that, despite the manual reviewing process and automatic testing of pull-requests, the overall quality of the repository (measured as the amount of uninstallable packages on any given switch) keeps decreasing, especially for older versions of OCaml, with over 20% of the packages not available for OCaml 3.12.1.

The OPAM Weather Service (OWS, http://ows.irill.org/), born on May 2014, provides a continuously updated HTML report on the repository status for major compiler versions, generated from the output of *distcheck*. The report tells, for each package, which of its versions can be installed for which of the compiler versions.

Figure 5 shows an excerpt of the report summary, available on the OWS main page. It shows two packages, both with multiple versions, and whether each of them is installable for 4 different versions of the OCaml compiler. This allows maintainers to check at a glance the health of a given package or version, each red box providing a link to a page explaining the reason why the package was found not to be installable.

Figure 6 shows such an explanation, for version 0.3.0 of package `mirage-www` on OCaml 4.01.0. The output shows a conjunction of dependency branches, leading to unsatisfiable constraints: unsatisfiable constraints usually sum up to either a


OCaml Version 4.01.0

Broken version mirage-www 0.3.0

The following dependencies couldn't be met:

- mirage-www → cstruct < 0.6.0
- mirage-www
  → mirage (≥ 0.8.0 & ≤ 0.9.0)
  → mirage-xen (= 0.9.8 | = 0.9.7 | = 0.9.6 | = 0.9.5 | = 0.9.4 | = 0.9.3 | = 0.9.2 | = 0.9.1)
  → cstruct ≥ 0.7.1
- mirage-www
  → mirage (≥ 0.8.0 & ≤ 0.9.0)
  → mirage-unix (= 0.9.8 | = 0.9.7 | = 0.9.6 | = 0.9.5 | = 0.9.4 | = 0.9.3 | = 0.9.2 | = 0.9.1)
  → cstruct ≥ 0.7.1
- mirage-www → mirage (≥ 0.8.0 & ≤ 0.9.0) → cstruct ≥ 0.6.0
- mirage-www → mirage-fs ≥ 0.4.0 → mirage ≥ 0.9.2 → cstruct ≥ 1.0.1
- mirage-www
  → mirage-fs ≥ 0.4.0
  → mirage ≥ 0.9.2
  → mirage-unix (= 0.9.8 | = 0.9.7 | = 0.9.6 | = 0.9.5 | = 0.9.4 | = 0.9.3 | = 0.9.2)
  → cstruct ≥ 0.7.1
- mirage-www
  → mirage-fs ≥ 0.4.0
  → mirage ≥ 0.9.2
  → mirage-xen (= 0.9.8 | = 0.9.7 | = 0.9.6 | = 0.9.5 | = 0.9.4 | = 0.9.3 | = 0.9.2)
  → cstruct ≥ 0.7.1
- mirage-www → mirage-fs ≥ 0.4.0 → cstruct ≥ 0.6.0

Here are the root causes:

- Unsatisfiable version constraints for cstruct
- Unsatisfiable version constraints for mirage

Fig. 6: An excerpt of `http://ows.irill.org/` showing the explanation given by the algorithm when a package version is not installable. Here, dependencies require that `cstruct` be older than 0.6.0 and more recent than 0.7.1 at the same time, and that `mirage` be older than 0.9.0 and more recent than 0.9.2 at the same time, giving two root causes for the problem.

missing version constraint (the package is not available at the required version), or conflicting version constraints (the set of versions for a dependency becomes empty). In this example, the two root causes are of the second kind, conflicting version constraints for both `cstruct` and `mirage`. Such information is quite useful to the maintainer: a missing version constraint may show that a package version was prematurely removed, leading to the re-addition of the package to solve the dependency; a conflicting version constraint has generally to be solved within the package itself, by increasing its compatibility with other packages, allowing to relax its version constraints.

Finally, it is interesting to study the impact of version management on the quality of the repository. Indeed, OPAM faces two more challenges, compared to other package managers: (1) each package can be available in multiple versions, and (2) the OCaml compiler itself is available in multiple – incompatible – versions, that may be installed at the same time in the user directory, in different *switches*.

Figure 7 plots the distribution of packages depending on the number of versions in which they are available. High numbers of packages versions are correlated with a lack of compatibility of each of them across compiler versions: it appears that fast moving developers tend to adopt new features of the language rapidly, not caring about compatibility. Pointing this out is a good way to push them towards maintaining backwards compatibility, and increasing the quality of the repository. This can be done either by proposing alternative implementations

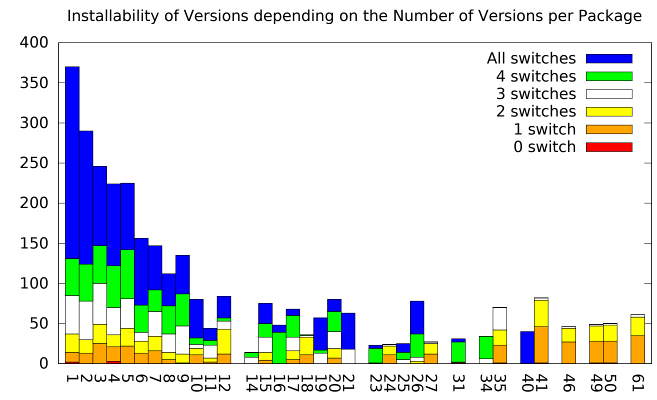


Fig. 7: We plot the number of versions per package (x), and the installability of each version (y), on the 5 different switches currently available. We can see that, in OPAM, many different versions of a package may be available at the same time (until 61), and that a big number of versions usually implies a worse compatibility between switches.

chosen at compile-time by a preprocessor, or by delaying the adoption of new language features until they have been available for a long enough time.

Figure 8 plots the installability of packages per OCaml version, while Figure 9 plots the evolution of installability for the specific 4.00.1 version of the compiler, which was born and replaced during OPAM lifetime. Both figures show that the community focuses its effort on the most recent versions of the OCaml compiler, increasing the compatibility of packages when the new version is released, and letting it go when a new version superseedes it.

Our findings show that the constant increase in the number of packages, package versions and compiler versions cause an amount of complexity that cannot be handled through manual review and installation testing.

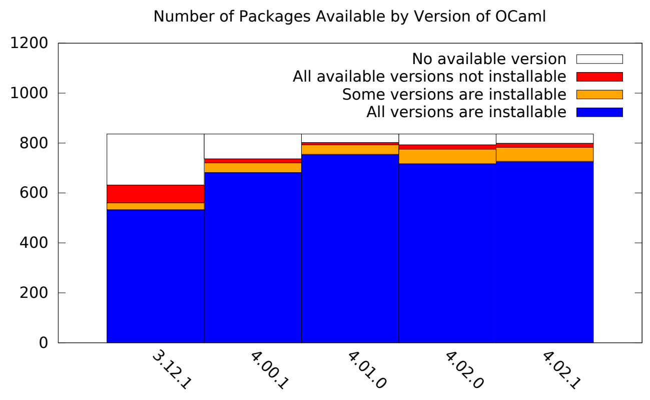


Fig. 8: We plot the installability of packages by OCaml version (switch). We see that, although 838 packages are available in the repository, more than 50 are not installable for each switch.

The deployment of the Opam Weather Service has already fostered healthy discussions among the OPAM community, and raised awareness of the need to use *distcheck* directly within the reviewing process of updates. In particular, tight integration with *distcheck* would allow maintainers to check the consequences of removing a package version, something that cannot be tested at all by the current testing process. It would also allow to focus backwards compatibility efforts on widely-used packages, which limit the available *switches* of all their dependents.

VI. The Drupal Contribution Repository

Drupal is a FOSS content-management framework written in PHP and distributed under the GPL license. It is used as a back-end framework for ≈2% of all web sites worldwide ranging from personal blogs to corporate, political, and government sites. The standard release of Drupal contains basic features common to content management systems. As of October 2014, there are more than 16'000 community-contributed modules (not considering themes), available to extend Drupal core capabilities, adding new features or customizing Drupal behaviour and appearance. Core and contributed modules are described using a standardized php-init file [6].

To apply our analysis to Drupal we first mirrored all modules from the Drupal Git repository; then, using a custom script, we have converted all modules metadata to CUDF. We created one universe for each Drupal release series, namely: 5.x, 6.x, 7.x. We did not consider the new upcoming Drupal release 8.x, which is in alpha stage at the time of this writing.

The metadata we used in this paper is partially extracted from the Drupal modules source (manually added by Drupal developers) and partially extracted from the Git repository.

[6]https://www.drupal.org/node/542202

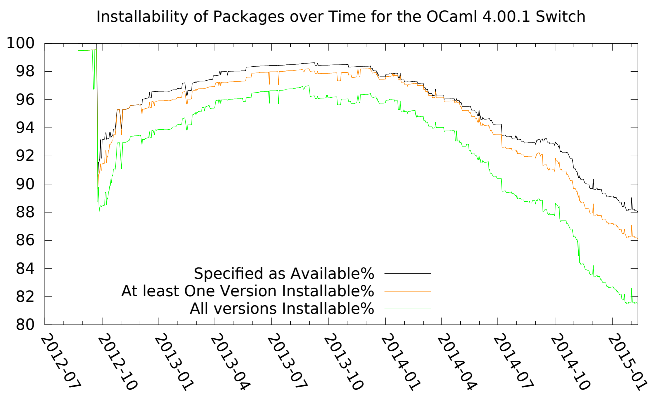


Fig. 9: We plot the evolution of installability over time for a specific version of OCaml, 4.00.1. Since the release of 4.00.1 in Sept 2012, installability increases until Sept. 2013, where a new version of OCaml is released. After that, a lot of packages are specified not to be compatible (above the black curve), which causes a lot more not to be (gap between the orange and black curves).

TABLE V: Drupal modules analysis

Release	Total	Broken	Unique Missing
5.x	17180	146	18
6.x	55814	412	80
7.x	93455	2064	190

Drupal module dependencies naturally map to conjunctions in CUDF. Since dependencies can have version constraints and no two versions of the same module can be installed at the same time, we made explicit this additional constraint via a self conflict: that is, a package is always in conflict between different module versions. Drupal metadata does not contain explicit conflicts.

The resulting CUDF universes for versions 5.x, 6.x and 7.x, contain respectively 16'000, 52'000, and 60'000 packages, accounting for all versions available for each Drupal release. We also considered 480 Drupal "distributions", that are specifically tailored collection of modules and themes.

The results of our analysis are pretty encouraging. Table V shows that despite the steady increase of modules in time, the number of broken packages is limited. Our investigation also shows that all installation problems are a result of missing dependencies, and the number of unique missing packages is a small fraction of the total number of modules. For example 161 modules are not installable because they require a module `field_collection_uuid` that at the time of writing has not been released and only available as a proposed patch [7].

VII. Discussion

We have applied *distcheck*, which is based on a well established sound and complete algorithm for detecting not installable packages [16] to 3 relevant software component repositories of different level of maturity.

On the Debian distribution, for which *distcheck* has been originally developed, our experience spans almost a decade since the installation of daily *distcheck* runs at the Debian QA meeting in December 2006. A detailed analysis of the last period of development of Debian has shown that *distcheck* was instrumental in improving the quality of the Debian distribution. In our opinion, a global quality assessment like the one done by *distcheck* can have a positive effect on the quality of a component repository when there are processes and tools that facilitate the resolution of the issues detected by analysis. In the case of Debian, an important tool is the *Bug Tracking System*[8] which keeps trace of open issues about individual packages. The rules governing packaging maintenance and bug resolution are documented in [4].

It is important to strike a balance between eagerness in error reporting, and avoiding to annoy package maintainers. This can best be achieved when one has internal knowledge of the culture of component maintenance that is behind the repository. Automated bug reports, for instance, are *a priori*

[7]https://www.drupal.org/node/2075325
[8]https://bugs.debian.org

perceived negatively in Debian, and are only accepted in restricted cases after prior discussion. When we started to systematically send bug reports on installability issues we therefore proposed precise criteria on when an issue merits a bug report, and announced our plans in advance on the developer mailing list [20].

We also always attempt to be as useful to package maintainers as possible when filing a bug. This includes checking whether a bug has already been filed, whether a different package is to blame for a non-installability issue, or whether an issue can simply be solved by recompilation of a source package. In the latter case, a recompilation request is sent to the team managing the autobuilder network, instead of a bug report against the package itself.

In the larger Debian ecosystem, *distcheck* has also been used by other distributions to *prevent* (as opposed to detect) installation issues to occur. In particular Emdebian,[9] a Debian-based distributions for embedded devices, has developed the `emdebcheck` wrapper around *distcheck* and has been using it since 2008. The key idea is that, *before* each package upload to the repository can happen, `emdebcheck` will simulate the new status of the remote repository as if the upload had already happened, run *distcheck* on it, and aborts the upload if installability issues are found. Theoretically, approaches like this one might guarantee that a given repository will never suffer of installability issues. In practice though, there are cases in which the distribution maintainers want to temporarily allow installability issues, for instance when bootstrapping new repositories; or when several upload of unrelated packages, possibly by unrelated developers, are needed to transition the repository from one steady state to another. Also, tightly controlled approaches like the Emdebian one are not necessarily portable to more complex infrastructure, where potentially long delays between developer upload time and the time the uploaded package is integrated into the repository might induce race conditions that invalidate the guarantee of absence of installability issues.

The OPAM package repository is a quite different environment: unlike Debian, it provides only source packages to its users, it keeps all old versions of a package in the same repository, and it is specialized for a single programming language, with *switches* intended to identify compiler versions instead of distribution releases. It is also a very young and dynamic repository sporting a fast growth rate. And yet, it turned out that applying *distcheck* to OPAM was relatively easy, with the major difficulty being the development of a new explanation engine, to account for the fact that multiple versions of a package are usually present in a repository.

Despite the fact that OPAM has a modern quality assurance process based on continuous integration tests, organized around GitHub pull requests, the historical analysis of the repository shows a very significant amount of non-installability problems, that keep growing. The OPAM weather service was developped to help keep these under control, and the maintainers have acknowledged that in needs to be included in its quality assurance process.

[9]http://www.emdebian.org/

Indeed, the OPAM weather service provides clear evidence that running intallation tests on the modified packages before accepting a pull request to the OPAM repository is not enough: work is in progress to include *distcheck* as a leading part in these tests, likely by comparing its results before and after the change, which gives a clear picture of its consequences to the quality of the repository (as opposed to the quality of a single package). Typically, if removing an older version of a package makes a whole part of the repository uninstallable, the maintainers should know.

Finally, our investigation of the Drupal module collections showed that *distcheck* can be applied even to component repositories with rather informal metadata, and despite the freshness of the results, that could not yet be confronted with feedback from the Drupal community, we could find a significant number of issues.

We believe that these findings provide very strong evidence of the relevance, applicability, and generality of repository quality checks based on the automated analysis of component interdependencies.

Hence, we do suggest that it would be highly beneficial to adopt tools like *distcheck* in all other component repositories where a clear, formalised quality assurance and release process are in place to include the results of the analysis, and provide proper response to the alarms raised. We hope that experience reports such as those presented here will help reach the necessary acceptance in the maintainer communities willing to incorporate this approach.

VIII. Threats to Validity

The data and results presented in this paper are based on metadata extracted directly from component repositories which are publicly available, and that contain metadata manually compiled by package maintainers in Debian, Drupal, and OPAM. The analysis is performed in all these cases by the same *distcheck* tool, which is based on a sound and complete algorithm organized around a pivot format, CUDF, which is formally defined and comes with a precise semantics.

Hence the main threat to the validity of our analyses come from the risk of errors introduced in the translation from the original metadata to CUDF.

This risk is limited for Debian: successive releases of *distcheck* have been in production use for almost 10 years now, and hence systematically exposed to the scrutiny of both Debian developers and. Also, important changes in the package metadata have been introduced only very recently (with the *multiarch* extension).

Since the OPAM package manager uses the CUDF format internally, we consider the risk of mistakes extremely limited, as the conversion is done by the OPAM developers themselves for their daily usage, and the community has hence performed an extensive validation of this conversion.

For the Drupal modules, despite these encouraging results, the validity of our study is limited by the quality of the metadata. The converter was developed by the authors, and the results have not been validated yet by the user community, so there is a risk that errors or misunderstanding in the conversion lead to wrong results. We had also to fix by hand a number of ill specified dependencies (mostly in the 5.x series) and human errors. The absence of explicit conflicts is also a limiting factor, as it is impossible for us to detect real conflict among modules (for example, modules that define the same menu entry in a website). We have also detected a few false positives due to the fact that we did not considered development releases.

Finally, we have a challenge related to *generalizability*: despite the fact that we have shown clear evident of the efficiency, extensibility and flexibility of our tool, *distcheck* may need improvements in the future to cope with the ever increasing size of component repositories, which is already in the range of several tens of thousand components. The explanation engine may also need to be adapted to continue providing clear and succinct explanations when multiple versions of the same component are present in the same repository (like in the case of OPAM).

IX. Related Work

A. Other applications of installability checks

As part of Debian QA efforts, *distcheck* is also being used for detecting cases of packages that fail at deployment time attempting to hijack files already present on the user machine, but owned by a different package [21]. This QA effort is performed in several steps. First we construct the set of pairs of packages in a repository that both claim ownership of a given file. Next, we use *distcheck* to restrict these to the pairs of packages that can be installed simultaneously, according to the model of Section II. This can be done in a single run of *distcheck* by constructing, for each pair of packages (a, b) obtained in the first step, a dummy package that depends on both a and b, and then checking for installability of all of these relative to the original repository. For all the pairs of packages that pass this second phase we have to test simultaneous installation in a test bed, since Debian provides several ways to patch the potential file conflict during deployment[10].

In February 2015, for the *unstable* distribution on the `amd64` architecture there are 859 pairs of packages that share a file (among the 10^9 possible pairs of packages), but only 72 can be installed simultaneously according to *distcheck*. Hence, using *distcheck* reduces the number of cases that need testing to less than 10% of the original search space.

[10]either in the metadata using a `Replaces` relation, or in installation scripts using the *diversion* mechanism

The detected bugs are tracked in the Debian bug tracking system[11]. We started this QA effort in April 2008 first by using an *ad hoc* solution for the final installation test. Today this is integrated in the *piuparts* test process [18]. By February 2015, a total of 743 bug reports of this kind were filed, 732 of which have been already resolved.

The *distcheck* tool is also being used to schedule package rebuilds within Debian in a way that is aware of unsatisfiable *build-time dependencies* (as opposed to deploy-time dependencies, which are the main concern of this paper). In particular, no build will be scheduled by the `wanna-build` scheduler,[12] unless *distcheck* can show that the build-time dependencies of a given *source* package are currently satisfiable in the origin repository. This way spurious build errors, e.g., due to *temporarily* unsatisfiable build-dependencies, are avoided, and the amount of human intervention needed to reschedule them at the right time is significantly reduced.

In [1], the temporal evolution of component repositories is formalized, and the notion of *outdated package* introduced. A package is outdated if not only it is not installable, but if it also requires changes in its own metadata before becoming installable again, i.e., no changes in *other packages* will suffice to make it installable. Outdated packages are a refinement of not installable packages discussed in this paper, and also important quality markers, because they also pinpoint *where* in the repository changes shall be made to correct installability issues.

B. Other approaches to non-installability checks

At the scale of component repository, an approach based on software product lines [10] has been proposed to, among other applications, detect not installable packages, i.e., empty software product lines. The approach is sound but Debian-specific, whereas *distcheck* works out of the box with metadata coming from different component models, have we have practically shown in this paper. Also, *distcheck* is much faster than typical SPL tools, which are usually applied to individual product lines that rarely have component numbers in the tens of thousand.

At the scale of individual component installation, on top of pre-existing user machine configurations, several works have studied the problem of why component upgrade fails and possible solutions to that problem. A good starting is [8], which reviews previous work in the area and propose a model to describe upgrade failures. As part of its analysis, the paper also shows how most failed upgrade causes are related to broken package dependencies. Although all non trivial component models require that users *might* be faced with component incompatibilities, the adoption of tools like *distcheck* guarantees that users will never be faced with *unsolvable* scenarios due to not installable packages.

[11]https://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=edos-file-overwrite;users=treinen@debian.org

[12]https://buildd.debian.org/

C. Dependency graph analysis

Several works have analyzed the dependency graph of large component repositories, including some of those we have studied in this paper. For instance [14], [15] have studied the Debian dependency graph to check whether interesting graph theoretical properties are verified or not. Those work are interesting in their own right, but do not help in verifying component installability, because they fail to take into account the underlying semantics of dependency graphs, which is rooted in their interpretation as propositional logic formulae.

Also, those studies do not take into account the temporal evolution of dependencies, like we have done in our Debian and OPAM case studies. Other studies have done so on other ecosystems though, like [5] in which the authors have studied the evolution of software dependencies (from the point of view of developer) in the Apache ecosystem over 14 years. As to the effects of introducing/upgrading dependencies, the paper focused on the amount of source-code-level changes induces by the change, rather than on the effect of package installability as seen at the inter-component relationship level.

X. Conclusion

In this paper we have presented our experience with the *distcheck* tool, which is specifically designed to identify all the non installable packages in a software repository. We have used *distcheck* in practice on three case studies involving large scale repositories coming from different communities and applicationd domains: Debian, OPAM, and Drupal.

Our findings show that *distcheck* produces efficiently precise and useful analysis of all noninstallable components found in all these repositories, providing precious information that can be used by component and repository maintainers to pinpointing and fix installability issues, a task which is otherwise highly nontrivial, due to the entanglement of dependencies and conflicts.

The result of these experiences, and the modular structure of the tool, that requires little effort to be extended for new kinds of component metadata, provide strong arguments for the adoption and integration of tools like *distcheck* into the quality assurance process of all component repositories.

References

[1] P. Abate, R. D. Cosmo, R. Treinen, and S. Zacchiroli. Learning from the future of component repositories. *Sci. Comput. Program.*, 90:93–115, 2014.

[2] P. Abate, R. di Cosmo, R. Treinen, and S. Zacchiroli. MPM: A modular package manager. In *CBSE 2011*. ACM, 21/06/2011 2011.

[3] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, October 2012.

[4] A. Barth, A. D. Carlo, R. Hertzog, L. Nussbaum, C. Schwarz, and I. Jackson. Debian developer's reference, version 3.4.14, 2014.

[5] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, pages 1–43, 2014.

[6] M. Caneill and S. Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *ESEM 2014: 8th International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014.

[7] E. Dolstra, A. Löh, and N. Pierron. Nixos: A purely functional linux distribution. *J. Funct. Program.*, 20(5-6):577–615, 2008.

[8] T. Dumitras and P. Narasimhan. Why do upgrades fail and what can we do about it? In J. Bacon and B. F. Cooper, editors, *Middleware 2009, ACM/IFIP/USENIX, 10th International Middleware Conference, Urbana, IL, USA, November 30 - December 4, 2009. Proceedings*, volume 5896 of *Lecture Notes in Computer Science*, pages 349–372. Springer, 2009.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[10] J. A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. Towards automated analysis. In *Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications*, pages 29–34. CEUR-WS.org, 2010.

[11] J. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[12] R. Hertzog and R. Mas. *The Debian Administrator's Handbook.* Freexian SARL, 2013.

[13] I. Jackson and C. Schwarz. Debian policy manual. http://www.debian.org/doc/debian-policy/, 2008.

[14] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[15] T. Maillart, D. Sornette, S. Spaeth, and G. von Krogh. Empirical tests of Zipf's law mechanism in open source linux distribution. *Phys. Rev. Lett.*, 101:218701, Nov 2008.

[16] 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: Automated Software Engineering*, pages 199–208. IEEE, 2006.

[17] R. Nguyen and R. C. Holt. Life and death of software packages: an evolutionary study of Debian. In *Center for Advanced Studies on Collaborative Research (CASCON)*, pages 192–204, 2012.

[18] L. Nussbaum. Use of grid computing for debian quality assurance. In *FOSDEM 2007, research room: Free and Open Source Software Developers' European Meeting*, 2007.

[19] L. Nussbaum and S. Zacchiroli. The ultimate Debian database: Consolidating bazaar metadata for quality assurance and data mining. In *MSR 2010: 7th IEEE Working Conference on Mining Software Repositories*, pages 52–61. IEEE, 2010.

[20] R. Treinen. mass bug filing: packages not installable on any architecture. https://lists.debian.org/debian-devel/2010/08/msg00063.html, Aug. 2010.

[21] R. Treinen and S. Zacchiroli. Solving package dependencies: from EDOS to Mancoosi. In *DebConf 8: proceedings of the 9th conference of the Debian project*, 2008.

[22] R. Treinen and S. Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project, Nov. 2009. http://www.mancoosi.org/reports/tr3.pdf.