# Bootstrapping Software Distributions*

Pietro Abate
Univ Paris Diderot, PPS,
UMR 7126, Paris, France
Pietro.Abate@pps.univ-paris-diderot.fr

Johannes Schauer
Jacobs University Bremen,
College Ring 3, MB670,
28759 Bremen
j.schauer@jacobs-university.de

## ABSTRACT

New hardware architectures and custom coprocessor extensions are introduced to the market on a regular basis. While it is relatively easy to port a proprietary software stack to a new platform, FOSS distributions face major challenges. Bootstrapping distributions proved to be a yearlong manual process in the past due to a large amount of dependency cycles which had to be broken by hand.

In this paper we propose an heuristic-based algorithm to remove build dependency cycles and to create a build order for automatically bootstrapping a binary based software distribution on a new platform.

## 1. INTRODUCTION

In recent years, the mobile and embedded device market have driven innovators to produce a large number of new devices and hardware platforms. In order to accelerate the adoption of a these new products, major commercial vendors chose to provide application developers with a platform agnostic virtual machine, taking the burden to *port* the native software stack to new architectures or hardware platforms. As a consequence, the efforts to adapt, to compile, and to fiddle with low-level details of the middleware components such as the kernel or the runtime system, are completely hidden from application developers. This model allows vendors to tightly control the number of dependencies among different software components and to reduce the porting procedure to a routine exercise.

The situation is different for collections of components based on Free and Open Source Software (FOSS). Recent efforts to provide a unified platform for mobile and desktop computing based on FOSS components (like Ubuntu for phones and tablets[1]) reignited the need to develop a model and tools to help distribution designers with the task to port, and natively compile software components to new platforms with minimal effort. In this context, a software distribution can be seen as a heavy middleware layer for the development of rich content platforms.

In FOSS distributions, components are developed independently by different communities and assembled together in software collections. Apart from intrinsic coordination problems associated to this distributed development model, the number of dependencies in FOSS distributions is an order of magnitude higher than what is found in proprietary systems, posing new challenges for automation and quality assurance. For example the number of vertices in the dependency graph of the android low level subsystem are less then a hundred, while a modern FOSS distribution is composed of more than ten thousand components. Porting a FOSS distribution to a new architecture not only involves adapting the low-level software layer for a different hardware, but also considering the inter-dependencies among different components and how these can affect the compilation and packaging process.

In our context the software components forming a distribution are called packages. We differentiate packages as *source* packages and *binary* packages. The first type are software units that can be compiled into binary form to produce a number of binary packages. The second are what are usually referred to as packages, that is, components that can be installed on an operating system. Both types use meta-data to describe their relationships to other components.

Source packages can also be seen as software product lines [8] where build dependencies identify specific configuration options to match the requirements of a family of software products. In the same optic, software distributions can be seen as software products customized for a specific device [9]. Bootstrapping a distribution to a new architecture deals with the problem of customizing a product line for a specific architecture and to instantiate a new software product that is consistent with the constraints imposed by the new hardware.

In this paper we will present algorithms and tools to help distribution architects with the task of analysing the dependency web associated to the compilation process. First we will provide a formal framework to reason about the bootstrapping problem (Section 2) and in Section 3 we will show few algorithms that we developed to untangle the dependency web. We provide a real application for these algorithms using data from the Debian distribution in Section 4 while we implementation details are presented in Section 5. In Section 6 we discuss related works in this area, and

*Work partially performed at IRILL center for Free Software Research and Innovation in Paris, France.

[1] `http://www.ubuntu.com/devices`


*CBSE 2013* Vancouver, Canada, June 17-21, 2013

we summarize our future plans and draw out conclusions in Section 7.

## 1.1 The Bootstrap Process

Bootstrapping a distribution is defined as the process by which software is compiled, assembled into deployable units and installed on a new device/architecture without the aid of any other pre-installed software. Different approaches are possible, such as using virtual environments emulating the target hardware, using a set of binaries often provided by the hardware vendor, or setting up an ad-hoc cross compiling environments on a machine with a different architecture. For example ARM routinely provides an emulator before the release of new hardware to ease its adoption, while cross compilation is mainly used by embedded distributions as the target hardware is often not powerful enough to compile software itself.

We define the architecture of the machine we compile *on*, the *native* architecture and the architecture we compile *for*, the *target* architecture. When native and target architecture are the same, the process to create a new software package is called *native compilation*; when they differ it is called *cross compilation*. The method presented in this paper involves, first the creation (by cross compilation) of a minimal build system, and later the creation of the final set of binary packages on the new device (by native compilation). Cross compiling a small subset of source packages is necessary because an *initial* minimal set of binary packages must exist in order to compile a larger set of source packages natively.

Once a sufficient number of source packages is cross compiled – we call the set of binary packages produced by them a *minimal system* – new source packages can be compiled natively on the target system. The minimal system is composed of a coherent set of binary packages that is able to boot on the new hardware and to provide a minimal working system. It contains at the very least an operating system, a user shell and a compiler and it is custom tailored by distribution architects. We developed tools to help them to refine and complete this selection. Even though the problems associated to cross and native compilation are conceptually similar, in this paper we will assume the existence of such minimal system and consider mainly the native compilation phase.

Native compilation entails computing an order in which source packages are compiled and the binary packages they produce are made available on the new system. This order is imposed by the presence of build dependencies, that is, binary packages that must be available on a system in order to create new binary packages from source packages. Source packages without build dependencies, or only with build dependencies that can already be satisfied on the minimal system can be immediately be compiled, and all the resulting binary packages made available. Source packages that require binary packages which are not yet available will be scheduled for later consideration.

In an ideal situation, where the build dependency graph entails a natural topological order, bootstrapping a distribution for a new architecture would not pose any particular challenge. Unfortunately, for larger distributions it is common that the build dependency graphs cannot be topologically sorted because the presence of millions of *dependency cycles*.

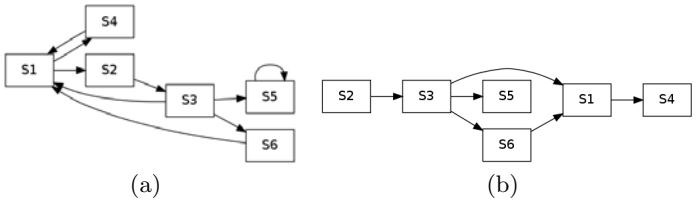



**Figure 1: Bootstrapping Problem Example**

In Figure 1 we give a simple artificial example where vertices represent source packages, and edges represent *dependencies* among source packages. For now, a dependency between two source packages $S_1, S_2$ is a connection defined by the fact that the package $S_1$, in order to be compiled, requires a binary package which will be provided by the source package $S_2$ only once $S_2$ is compiled.

In Figure 1(a) we can identify four cycles. The self cycle of vertex $S_5$ is a typical example of a cycle associated with the source package of a programming language that requires itself to compile. Other cycles have length 2, 3 and 4. Even if there are multiple ways of removing dependency cycles, it is not possible to develop an exact algorithm by looking at the packages' meta-data. Up until now, whenever the distribution is to be bootstrapped, package maintainers inspected source packages by hand to decide which build dependencies are really needed and which are optional. We developed a few heuristics to help maintainers with the task to identify such build dependencies. In Figure 1(b) we are able to obtain a directed acyclic graph (DAG) by removing the dependencies of $S_4$ on $S_1$, $S_1$ on $S_2$ and the self-cycle of $S_5$ with itself. From it we can easily compute a build order where package $S_4$ and $S_5$ are compiled first (possibly in parallel), then package $S_1$, followed by $S_6$ and finally $S_3$ and $S_2$.

Once all source packages are compiled, we can restart the process, but now re-instantiating all the build dependencies previously dropped. In this second compilation stage, all the features that were disabled to remove build cycles, are now enabled once again. Since all binary packages have now been compiled, the build dependencies of all source packages are satisfied and they can therefore be compiled in any order. This recompilation is necessary to build every component with its full feature set.

### *Our Contribution.*

By analysing the nature of these cycles and by pin-pointing packages that are more relevant then others we provide heuristics and tools to help developers and distribution editors to reduce the burden associated to the bootstrapping process and to reduce the time to market of a software distribution on a new platform. We present an heuristic-based algorithm to break dependency cycles and we validate our results using the Debian/Gnu Linux distribution as an example.

# 2. DEFINITIONS

In this section we will introduce a formal framework to model the bootstrap problem. First we will define package repositories and the notion of installability [15]. Then we will explain how mixed repositories, composed of source and binary packages can be used in order to compute a compi-

```
Package: S1
Depends: a, b
Binaries: d, e

Package: S2
Depends: a, d | e
Binaries: a

Package: S3
Binaries: b

Package: S4
Binaries: c
```

```
Package: a

Package: b
Depends: c

Package: c

Package: d
Depends: a

Package: e
Conflicts: a
```

**Figure 2: Binary/source package repository**

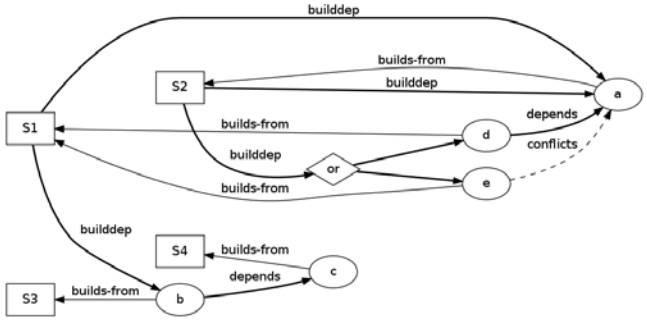



**Figure 3: A dependency graph with source and binary packages.**

lation plan and to create a set of binary packages from a set of source packages. Finally we will give a formal definition of the bootstrap problem. We will use the set of packages in Figure 2 (graphically presented in Figure 3) as a running example. Rectangles and ellipses represent source and binary packages, dependencies and conflicts are drawn with bold and dashed arrows, dependency disjunctions are split by a rhombic shape and the relation between binary packages and source packages is represented as a light arrow.

### *Packages.*

A package is a tuple $(n, v)$ where $n$ is a package name and $v$ is a version. Package names are arbitrary strings, and we assume that versions are non-negative integers.

DEFINITION 2.1. *A repository is a tuple* $(P, Dep, Con, Bin)$ *where*

- $P$ *is a set of packages*
- $Dep : P \to \mathcal{P}(\mathcal{P}(P))$ *is the dependency function (we write* $\mathcal{P}(X)$ *for the set of subsets of* $X$*)*
- $Con \subseteq P \times P$ *is the conflict relation*
- $Bin : P \to \mathcal{P}(P)$ *is the binary function*

The *binary* function $Bin$ is used to associate source packages to binary packages. If $Bin(p)$ is the empty set, then $p$ is identified as a binary package.

DEFINITION 2.2. *Let* $(P, Dep, Con, Bin)$ *be a repository and let* $p \in P$*. If* $Bin(p) = \emptyset$ *then we call* $p$ *a* binary package *else we call* $p$ *a* source *package.*

In Figure 2, packages $a, b, c, d, e$ are binary packages, while $S1, S2, S3, S4$ are source packages. The field `Depends` is given as a conjunction of disjunctions, i.e. $Dep(S2) = \{\{a\}, \{d, e\}\}$.

DEFINITION 2.3. *Let* $R = (P, Dep, Con, Bin)$ *be a repository and* $B \subseteq R$ *the subset of binary packages in* $R$*. Then the repository must satisfy the following conditions:*

- *The relation* $Con$ *is symmetric i.e.,* $(p, q) \in Con$ *if and only if* $(q, p) \in Con$ *for all* $p, q \in P$*.*
- *For all* $p \in P$ *,* $Bin(p) \subseteq B$*.*
- *Neither binary nor source packages can depend on source packages nor can they conflict with source packages. For all* $p \in P$ *,* $Dep(p) \subseteq B$ *and* $Con(p) \subseteq B$

The dependencies of a binary package indicate which packages must be installed together with it, the conflicts which packages must not. The *build* dependencies of a source package identify binary packages that must be installed in order for the source package to be compiled. *Build* conflicts are binary packages that must not be installed in order to compile the source package. As by Definition 2.3, the build dependencies of source packages are always binary packages.

DEFINITION 2.4. *Let* $(P, Dep, Con, Bin)$ *be a repository. The dependency relation is a binary relation* $\hookrightarrow$*:* $P \times P$ *defined as* $p$ depends on *a package* $q$ *if there exists* $D \in Dep(p)$ *such that* $q \in D$*.*

This definition can be extended to a multi-step relation: $q$ is in the *dependency closure* of a package $p$ if $p \hookrightarrow q$ in one ore more steps. We then say $p \hookrightarrow^+ q$. In Figure 2, the dependency closure of the package $S1$ is the set $\{a, b, c\}$. While packages $a, b$ are direct dependencies, package $c$ is a transitive dependency.

DEFINITION 2.5. *Let* $(P, Dep, Con, Bin)$ *be a repository* $p \in P$*. The dependency closure of* $p$ *is a subset of* $P$ *defined as*

$$\{q \mid q \in P, p \hookrightarrow^+ q\}$$

Source packages and binary packages are related. A binary package is *built from* a source package, and a source package *builds* a set of binary packages.

DEFINITION 2.6. *Let* $(P, Dep, Con, Bin)$ *be a repository. The function* $Src : P \to P$ *is defined as follows:*

$$Src(p) = s \text{ such that } p \in Bin(s)$$

In Figure 2, the source package corresponding to the binary package $a$ is $S2$.

### *Installation Sets.*

An installation is a consistent set of packages, that is, a set of packages satisfying abundance (every package in the installation has its dependencies satisfied) and peace (no two packages in the installation are in conflict) [15]. Formally:

DEFINITION 2.7. *Let* $R = (P, Dep, Con, Bin)$ *be a repository. An* $R$-installation $I$ *is a subset* $I \subseteq P$ *such that for every* $p \in I$ *the following properties hold:*

**Abundance.** *Every package has what it needs: for every $p \in I$, and for every dependency $D \in Dep(p)$ we have $I \cap D \neq \emptyset$.*

**Peace.** *No two packages conflict: $(I \times I) \cap Con = \emptyset$.*

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

DEFINITION 2.8. *Given a repository $R = (P, Dep, Con, Bin)$ and a package $p \in P$. We define the partial function $IS_R(p) = I$ to denote one $R$-installation such that $p \in I$.*

We note that in Figure 2, even though the dependency of the package $S2$ contains a disjunction, the installation set of $S2$ is unique, namely $IS(S2) = \{a, d\}$.

DEFINITION 2.9. *Let $R = (P, Dep, Con, Bin)$ be a repository and $B = \{p \mid Bin(p) = \emptyset\}$. We say that $R$ is* self-contained *if the following conditions hold:*

$\forall b \in B\ ,\ Src(b) \in S$
$\forall s \in S\ \exists\ I \subseteq B$ *such that* $I \cup \{s\}$ *is a* $R$*-installation*

In other words a repository $R$ is self-contained if all binary packages in $R$ are built from the source packages in $R$ and all source packages in $R$ can be built using only binary packages in $R$.

LEMMA 2.10. *Let $R$ be a self-contained repository. Then all source packages in $R$ can be compiled.*

### *Compilation.*

The notion of *R-installation* applies to all types of packages. Intuitively, for a package $p$ there exists an *R-installation* if all (runtime or build) dependencies can be satisfied. However, in order to characterize the compilation process we need to further introduce the notion of *R-compilation*. Intuitively a source package $s$ can be compiled in a repository $R$ if there exists an *R-installation* $I$ and $s \in I$ and all source packages used to build binary packages in $I$ can also be compiled. The compilation of a package, not only implies that there exists an installation set $I$ for its build dependencies, but also that all source packages associated to all binary packages in $I$ can be also compiled.

First we define the following relation among source packages.

DEFINITION 2.11. *Let $(P, Dep, Con, Bin)$ be a repository. The binary relation $\sim$: $S \times S$ is defined as $s \sim t$ if there exist a installation set $I$ such that $s \in I$ and some $p \in I$ , $t = Src(p)$.*

We now extend the one step definition above to an arbitrary number of dependency steps.

DEFINITION 2.12. *The relation $\sim^*$: $S \times S$ is defined as $s \sim^* t$ if: $s \sim t$ or there exists $s_1 \cdots s_n$ such that $s \sim s_1$, $s_n \sim t$ and $s_i \sim s_{i+1}$ for $1 \leq i \leq n$.*

The relation $\sim^*$ is transitive. We define an *R-compilation* of a source packages $s$ as the closure of $P$ under $\sim^*$.

DEFINITION 2.13. *Let $R = (P, Dep, Con, Bin)$ be a repository and $s$ a source package. An* $R$-compilation *of $s$ is a set $I \subseteq P$ such that $I = \{t \mid s \sim^* t\}$. We say that a source package $s \in S$ can be compiled if there exists an* $R$-compilation *$I$ such that $s \in I$.*

For the repository in Figure 2, the *R-compilation* set of the package $S1$ is the set $\{S1, S2, S3, S4\}$: to compile the package $S1$ we need the binary packages $a, b$ and transitively package $c$. The package $a$ is generated by the source package $S2$, while $Src(b) = S3$ and $Src(c) = S4$. Since $S3$ and $S4$ do not have any build dependencies, they do not pose any problem. However, it is easy to notice that the package $S2$ build-depends also on the package $d$ that is in turn generated by the source $S1$ giving an example of a build dependency cycle.

Since an *R-compilation* for a given package is not unique, we identify a unique subset of packages common to all *R-compilation*s. This subset is identified restraining Definition 2.11 to strong dependencies [1]. We quickly recall the definition:

DEFINITION 2.14. *Given a repository $R$, we say that a package $p$ in $R$* strongly depends *on a package $q$, written $p \Rightarrow q$, if $p$ is installable in $R$ and every installation of $R$ containing $p$ also contains $q$.*

Intuitively, $p$ strongly depends on $q$ with respect to $R$ if it is not possible to install $p$ without also installing $q$. In Figure 2, the package $S2$ strongly depends on the packages $d$ and $a$: the package $a$ is a direct dependency, while the package $d$, even if part of a disjunctive dependency, is the only choice because of the conflict between packages $d$ and $a$.

DEFINITION 2.15. *Let $R$ be a repository. Two source packages are related $s \simeq t$ if there exist $I = \{q \in R \mid q \Rightarrow s\}$ such that $s \in I$ and for some $p \in I$ , $t = Src(p)$.*

In the same way as in Definition 2.12 we can define $\simeq^*$ as the closure of the relation $\simeq$ and finally define the *Core R-compilation* as in Definition 2.13, but w.r.t. the relation $\simeq^*$. The *Core R-compilation* allows us to focus on a unique subset of *R-compilation* sets and to provide a lower limit to the dependencies needed in order to compile a source package.

If an *R-compilation* set exists, then it is a-priori not unique. This problem is due to the fact that package dependencies contain disjunctions. However, because of the specific nature of the problem, in practice, this problem does not arise too often: The number of disjunctions in build dependencies are often very limited. Moreover, using specialized solvers [2], we can use heuristic to minimize the number of packages to consider, possibly reducing the number of build cycles. The restriction to *core R-compilation* sets allow us to provide a lower bound of the problem.

### *Source Graph.*

We define the source graph associated to an *R-compilation* set as follows:

DEFINITION 2.16. *Let $R = (P, Dep, Con, Bin)$ be a self-contained repository and $I$ be an* $R$-compilation *set for a set of source packages $S \subseteq P$. The associated source graph $G = (V, E)$ is defined as*

- $V = I$
- $E = \{(s, t) \mid s, t \in I \ \text{ and } \ s \sim t\}$.

Figure 4(a) shows the source graph associated to the repository in Figure 2. In the same way we can define the *core*

source graph of a *core R-compilation* set by using the $\simeq$ relationship instead. The core source graph is a proper subgraph of every source graph no matter which *R-compilation* set is chosen.

Note that the source graph might contain cycles and hence it is not possible to use a topological sort algorithm. In the next section we will provide methods to compute the set of packages that can be compiled in a repository and heuristics to drop optional build dependencies from the source graph and to generate a build order.

When we mention build dependency cycles, then we imply the cycles to be elementary:

DEFINITION 2.17. *Given a graph* $(V, E)$*, a* path *is a sequence of vertices* $v_1 \cdots v_n$ *such that for each* $i, 0 \leq i \leq n-1$ *,* $(v_i, v_{i+1}) \in E$*, that is, between every subsequent pair of vertices there is an edge connecting them. If* $v_1 = v_n$ *then a path is called a* cycle*. A cycle is called* elementary *if no vertex (except the first and last) appears more than once in it.*

## 2.1 Problem statement

To formally describe the bootstrap problem, we first need to define the tools to relax build dependency constraints of source packages to, in the end, be able to compile all source packages in a given repository. A *build profile* is a function which transforms repositories to make the source graph acyclic. Formally:

DEFINITION 2.18. *A build profile Pmap is a function that transform repositories into repositories.*

*Let* $R = (P, Dep, Con, Bin)$ *and* $R_1 = (P, Dep_1, Con_1, Bin_1)$ *such that* $R_1 = Pmap(R)$*.* $R_1$ *must satisfy the follows constraints:*

- $Bin_1 \subseteq Bin$*.*
- *If* $R$ *is self-contained then* $R_1$ *is self-contained.*
- *Let* $G$ *and* $G_1$ *be the source graphs associated respectively to* $R$ *and* $R_1$*. Then the number of elementary cycles of* $G_1$ *is less or equal then the number of cycles in* $G$*.*

Finally we can give the formal problem statement. The bootstrap problem is defined as a sequence of refinements where, starting from a repository $R = (P, Dep, Con, Bin)$ and a minimal build system $B_0 \subset P$, all source packages in $R$ are compiled throughout multiple iterations, increasing the amount of binary packages $B_0 \cdots B_n$ until all source packages can be compiled and produce all binary packages $B$ in the repository $R$. Formally:

DEFINITION 2.19. *Given a repository* $R = (P, Dep, Con, Bin)$*, and a set of binary package* $B_0 \subset P$*, the bootstrap problem*

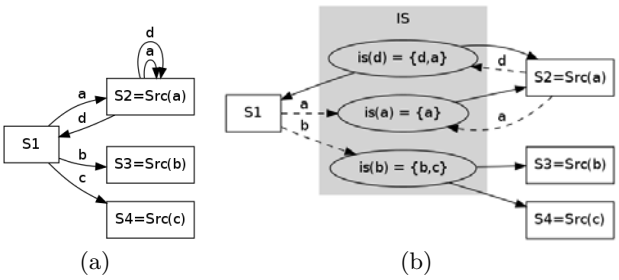



**Figure 4: A source graph 4(a) and build graph 4(b) example**

*is defined as a sequence of build profiles such that:*

$$
\begin{aligned}
R_1 &= Pmap_1(R) \\
S_1 &= Src(R_1) \\
B_1 &= Bin(Installable(S_1, B_0))
\end{aligned}
$$

$$
\begin{aligned}
R_2 &= Pmap_1(R_1) \\
S_2 &= Src(R_1) \\
B_2 &= Bin(Installable(S_2, B_1))
\end{aligned}
$$

$$\ldots$$

$$
\begin{aligned}
R_n &= Pmap_n(R_{n-1}) \\
S_n &= Src(R_n) \\
B_n &= Bin(Installable(S, B_{n-1})) = B
\end{aligned}
$$

*and for all* $i, j$ $R_i \neq R_j$*.*

The function $Installable(S, B)$ computes the set of packages $S_i \subseteq S$ that are compilable in the repository $S \cup B$ as by definition 2.7 of an $R$-installation.

$B_0 \cdots B_n$ are sets of binary packages at each step $i$ can be used to resolve the dependencies to compile the source package in $S_i$. The set $B_0$ is the minimal build system. The last step makes the packages in $B_n$ available which is equal to the set of binary packages $B$ in $R$.

Since at each iteration the set $B_i$ grows, less and less source packages will require to be modified by changing their build dependencies, finally leading to the original set of source packages $S$ to be compiled. The result of this compilation will be the original set of binary packages $B$.

# 3. ALGORITHMS

One important application associated with this work is the development of an automatic build procedure, that will be used to compile, test and assemble packages for different architectures. An essential building block toward this goal is the development of heuristics and tools to create an appropriate build order to guide such infrastructure.

To compute such order we first need to arrange packages in a graph and then transform it using ad-hoc heuristics to remove all eventual cycles. The final result will be then obtained by topological ordering the vertices in the resulting graph. In Section 3.3 we will also present two algorithms used to select a coherent subset of a repository based on a user specification.

## 3.1 Build Dependency Graph

In Definition 2.16 we introduced the source graph as a tool to reason about build dependency cycles. However, for efficiency reasons, in our implementation, we use instead an intermediary data structure that we call the *build graph*. This data structure embeds more information than a source graph and can be easily converted into a source graph. The build graph will be used as input for the edge removal algorithm in Section 3.2.

Directly using a source graph has the disadvantage of creating one edge for each binary package in the installation set (Figure 4(a)) making the edge removal procedure an expensive operation. The build graph obviates this problem by introducing an intermediary vertex between source vertices. We call this new vertex kind *installation set vertices*. We call an edge between a source vertex and an installation set vertex a *build-depends* edge, while we call an edge between an installation set vertex and a source vertex a *builds-from* edge. In Figure 4(b), the former is drawn as a dashed line and the latter as a solid line. Rectangles represent source package vertices, ellipses represent installation set vertices.

For example, Figure 4(a) shows the source graph of the repository in Figure 2, where $S1$ build depends on the binary packages $a$ and $b$. Besides edges for those two binary packages, the source graph also contains an edges for the binary package $c$. This is because $c$ happens to be in the installation set of $b$ and is therefore also in the installation set of $S1$. It is not obvious from this representation how the source graph should be modified if either of the two build dependencies of $S1$ ($a$ or $b$) would be dropped. Figure 4(b) shows the corresponding build graph. $S1$ now no longer has an edge for $c$ but instead, $c$ is part of the installation set of $b$ to which $S1$ connects. Using this representation, it can immediately be seen how its connection to $S2$ and $S3$ would be severed if its dependency $b$ was dropped.

**Algorithm 1** Build Graph Algorithm

```
1: procedure BUILD_GRAPH(S, B, R)
2:     for all s ∈ S do
3:         I ← IS_R(s)
4:         P ←PARTITION(I, Dep(s))
5:         for all is ∈ P do
6:             if is ⊈ B then
7:                 ADD_EDGE(s, is)
8:                 for all b ∈ is do
9:                     if b ∉ B then
10:                        t ← Src(b)
11:                        ADD_EDGE(is, t)
```

Algorithm 1 describes the creation of the build graph. The function BUILD_GRAPH takes as arguments a set of source packages $S$, a set of binary packages $B$ and a repository $R$ such that $S, B \subseteq R$. The set $B$ holds all the packages that should not be included in the build graph. First, for each source package $s$, we compute an $R$-installation set $I$ and we partition it into subsets. The installation set $I$ is usually small and it is computed using a specialized sat solver [2]. The set $P$ is a set of sets defined as follows

$$P = \{DependencyClosure(p) \cap I \mid p \in Dep(s)\}$$

and computed by the function PARTITION. We then add edges from $s$ to all installation set partitions $is$. If the installation set associated to a build dependency is a subset of $B$ then the subgraph associated to this dependency can be omitted. In the end, we add edges to all source package vertices $t$ that build the binary packages $b$ in the installation set $is$. The set $B$ contains all those binary packages that are not relevant to the construction of the build graph because they are part of the initial minimal build system or have been cross or natively compiled earlier.

Consider the repository $R$ of Figure 2. Starting from the source package $S1$, the algorithm proceeds as follows: The direct dependencies of $S1$ are $Dep(S1) = \{a, b\}$ and the $R$-installation of $S1$ is $IS_R(S1) = \{a, b, c\} = I$. The function PARTITION$(I, Dep(S1))$ in this case returns $\{\{a\}, \{b, c\}\}$. Two installation set vertices are created containing those two sets respectively. The vertex for $S1$ is connected to each of them using build-depends edges. As the last step, builds-from edges are added to both installation set vertices. They point to the source package vertices that the binary packages $a$, $b$ and $c$ build from which are $S2$, $S3$ and $S4$ respectively. Those three source packages are then processed in the same fashion and the result of the computation can be seen in Figure 4(b).

REMARK 3.1. *The Algorithm 1 is not complete. Since the installation set computed at each step is not unique there exist multiple possibilities for a build graph involving the same initial source packages. By considering edges that correspond to strong dependencies as in Definition 2.14 we can create a sub-graph which only consists of vertices which are present in every possible selection of installation sets.*

### *Source Graph.*

The source graph, as defined in Definition 2.16, is computed by path contraction from the build graph over all builds-from edges. As a result of the contractions, all installation sets vertices are removed from the graph (Figure 4(a)). This operation can be done in $O(n + m)$ with $n$ and $m$ being the number of vertices and edges in the build graph respectively.

## 3.2 Finding Build Profiles

Since the build graph (and thereby the source graph) may contain dependency cycles, in this section we present three heuristics to identify a "minimal" set of source packages that, by relaxing their dependencies, will make the build graph acyclic. We start by identifying all cycles of length 2. Removing *all* such cycles is a pre-requisite to transform the build graph into a DAG. Then we introduce a simple heuristic based on local vertex characteristics to automatically identify candidate edges to be removed. The last heuristic provides a way to deal with cycles of arbitrary length. These heuristics are meant to provide packages maintainers with tools to highlight important packages and recurrent patterns hidden in the build graph.

### *Removing 2-cycles.*

Dependency cycles of length two are most often encountered for source packages of programming languages which need themselves to be compiled (Vala, Python, SML, Free Pascal, Common Lisp, Haskell). In a build graph those are identified by a sequence of one build-depends edge and one builds-from edge in the opposite direction. They contain exactly one source vertex and one installation set vertex. Since there is only one way to break a 2-cycle (only build-depends

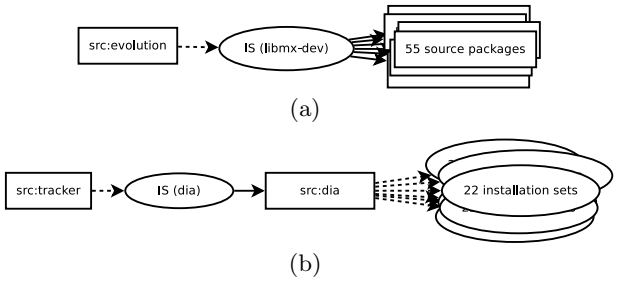



**Figure 5: Two in-/out-degree based ratio heuristic examples**

edges can be removed), we simply enumerate all 2-cycles to create a list a list of edges that we *must* deal with to transform the build graph in a DAG. In Figure 4(b), there exist two 2-cycles between $S2$ and $is(d)$ and $S2$ and $is(a)$. The only way to break them is to remove the build dependencies $a$ and $d$ of $S2$. When a cycle cannot be broken because the dependency is indeed not optional, the solution is to cross compile the source package or the set of source packages which generates the binaries in the installation set. Moreover, we identify all 2-cycle in the *core* build graph which is constructed considering only strong dependencies (Definition 2.14). This will provide a lower-bound to the 2-cycles that must be removed.

### *Relaxing dependencies using vertex based heuristics.*

We provide two metrics to identify source packages or dependencies that may heavily impact the compilation of another source package. Removing these "heavy dependencies", often reduce the number of cycles in the graph. The basic reasoning behind both heuristics is that often FOSS software depends on large software packages to borrow a small number of features. By removing these dependencies, the package will be successfully compiled with a reduced set of options without compromising its core functionalities.

Figure 5(a) displays a situation where a heavy dependency will imply 55 more source packages to be compiled. In this case, if `evolution` can be compiled without the binary package `libmx-dev`, then its connection to 55 other source packages is severed and the build graph will be considerably simplified.

Figure 5(b) displays a similar situation where one build dependency proxies the connection to a heavy source package that will require a large number of dependencies to be satisfied. In this case, the source package `src:dia` only has one predecessor installation set of `dia` which in turn only has one predecessor source package `src:tracker`. So if `src:tracker` can be compiled without `dia`, then `src:dia` can be removed from the graph together with all its connections to 22 other installation sets.

Another minor heuristic is based on the experience gained during the years by distribution architects. In particular, it is common knowledge that functional packages used to generate documentation or to run unit tests suite are not essential for the functionality of a package. By identifying and removing those dependencies, it is possible to further simplify the build graph.

### *Cycle based heuristics.*

**Algorithm 2** Approximate feedback arc set algorithm

**procedure** PARTIALFAS($C_i, A_i$)
  **if** $C_i = \emptyset$ **then**
    **return** $(C_i, A_i)$
  **else**
    $e \leftarrow$EDGEWITHMOSTCYCLES($C_i$)
    $C_{i+1} \leftarrow C_i \backslash$CYCLESTHROUGHEDGE($e$)
    $A_{i+1} \leftarrow A_i \cup \{e\}$
    **return** PARTIALFAS($C_{i+1}, A_{i+1}$)
**procedure** RECCYCLE($G_i, A_i, n$)
  **if** HASCYCLE($G_i$) **then**
    $C \leftarrow$FINDCYCLES($G_i, n$)
    **if** $C \neq \emptyset$ **then**
      $A \leftarrow$PARTIALFAS($C, \emptyset, \emptyset$)
      $G_{i+1} \leftarrow$REMOVEEDGES($G_i, A$)
      $A_{i+1} \leftarrow A_i \cup A$
      **return** RECCYCLE($G_{i+1}, A_{i+1}, n+2$)
    **else**
      **return** RECCYCLE($G_i, A_i, n+2$)
  **else**
    **return** $A_i$
$findFAS \leftarrow$RECCYCLE($G, \emptyset, n$)

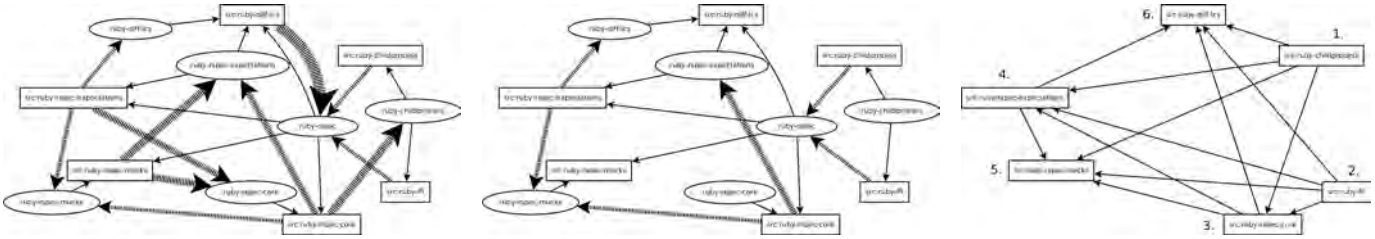
The two previous heuristics consider local vertex attributes. Now we present an algorithm to analyse large cycles that will be otherwise very complicated to see by direct inspection.

We use a modified version of Johnson's algorithm [12] to finds all elementary cycles up to a given length. Johnson's algorithm has a complexity $O((v+e)*n)$ where $n$ are the number of cycles in the graph and, $v, e$ are respectively the number of vertices and the number of edges. Our algorithm has a similar complexity, but $n$ is bounded.

We present an approximate solution to the feedback arc set problem. A *minimal* feedback arc set is the smallest (possibly not unique) set of edges which, when removed from the graph, makes the graph acyclic. Since the feedback arc set problem is NP-hard [13] and our graph contains components with up to a thousand vertices, it is computationally not feasible to identify an optimal solution. The proposed algorithm is based on the consideration that at least one edge from every cycle in the graph must be part of a feedback arc set and that those edges that are common to a large number of cycles are best candidates to be removed. The output is a set of build dependencies which, if dropped, makes the build graph acyclic.

We use a greedy strategy which at first enumerates all elementary cycles up to a specific length and then iteratively tries to remove the edges with most cycles through them. This step is repeated increasing the length of the largest cycles to consider until the graph is cycle free.

Algorithm 2 is composed of two recursive functions and takes as input a strongly connected subgraph $G$ of the build graph and an initial cycle length $n$. It returns a set of build-depends edges which, if dropped, turn the strongly connected component into a directed acyclic graph. The function RECCYCLES first checks if the input graph is cycle free and in this case it returns the feedback arc set. Otherwise it first computes the list of cycles of length $n$ to be analysed with the function PARTIALFAS. The latter greedily removes edges that are part of most cycles until all are broken. Once all edges found by this method are removed,



(a) (b) (c)

**Figure 6: Strongly connected component 6(a), a directed acyclic build graph 6(b), a directed acyclic source graph and a build order 6(c)**

we can then use a standard topological ordering procedure to extract a build order.

### *Edge Removal Example.*

In Figure 6 we describe the execution of the procedure used to remove cyclic dependencies from a build graph using algorithms and heuristics presented so far in this section. Figure 6(a) shows a strongly connected component computed from the build graph of the Debian unstable distribution using the Algorithm 1. Dashed edges are build-depends edges and their width represents the amount of cycles through them computed during the execution of Algorithm 2. Figure 6(b) shows the result of applying a build profiles set (Algorithm 2) to the build graph where the edges with most cycles through them are removed. First all 2-cycles are removed. Then we enumerate larger cycles (ex. `src:ruby-diff-ics`, `ruby-rspec`, `src:ruby-rspec-core`, `ruby-rspec-expectations`, `src:ruby-diff-ics`) and we apply the function `EdgeWithMostCycles` in Algorithm 2. As a result the build dependency of the source package `src:ruby-diff-ics` on the binary package `ruby-rspec` is severed as well as other *heavy edges*. Figure 6(c) shows the associated source graph obtained by path contraction (Section 3.1) from the build graph in Figure 6(b). Numbers represent the build order obtained by sorting the vertices topologically.

REMARK 3.2. *In Algorithm 2, the ratio between execution time and quality of the result can be adjusted setting the initial maximum length of cycles to enumerate. Our results show that the gain obtained considering cycle lengths larger the 10 is small compared to the time needed to compute longer cycles.*

## 3.3 Minimal Build Sets

In this section we present two algorithms that are used to compute a "reduced" distribution that is self-contained (Definition 2.9) and large enough to be representative of the entire distribution, but with less noise. In the context of heuristic based algorithms presented in this paper, it is important to focus only on the core part of the compilation problem. A reduced distribution allows to focus on a meaningful, small and manageable subset of packages. Given an initial set of source packages, the following two algorithms compute respectively the maximum amount of source packages which can be compiled from an initial set of binary packages and the minimum amount of source packages needed to create a self-contained repository. The *compilation fix-point* procedure is used in the execution pipeline (Section 5) to identify the largest set of source packages that is possible to compile from the minimal build system. The *build-closure* algorithm is used to select the smaller number of packages needed to compile a set of binary packages in order to reduce the number of possible build cycles.

### *Compilation Fix-Point.*

We use this algorithm to identify source packages in $S$ which can be compiled from a given set of binary packages $M$ (for example the minimal build system) without relaxing the build dependencies of any source package. The result of the algorithm is a tuple $(B, C)$ of binary and source packages where the set $B$ then can be used in the Algorithm 1 to exclude packages from the build graph.

**Algorithm 3** Compilation Fix Point

```
1: procedure F(B_i, C_i, S_i)
2:     NS ← Installable(S_i, B_i)
3:     if NS = ∅ then
4:         return (B_i, C_i)
5:     else
6:         B_{i+1} ← Bin(NS) ∪ B_i
7:         C_{i+1} ← C_i ∪ NS
8:         S_{i+1} ← S_i \ NS
9:         return F(C_{i+1}, B_{i+1}, S_{i+1})
10: Fixpoint ←F(M, ∅, S)
```

The Algorithm 3 proceeds as follows: first we compute the set of all source packages that can be compiled in the given repository composed by the binary packages in $B_i$. If such set is empty then we return the set of binary packages and the set of source packages. Otherwise we create three new sets, $B_{i+1}, C_{i+1}$ and $S_{i+1}$, respectively the set of binary packages built so far, the set of source packages compiled so far and the set of source packages that are left to compile. We repeat this function until no more source packages can be compiled.

### *Build Closure.*

The build closure algorithm is used to compute the minimum amount of source packages needed to create a self-contained repository as in Definition 2.9.

First we compute the union of all installation sets for all source packages in $S$: $NB = \bigcup_{s \in S} IS(s)$. If $NB$ is empty then it means that either $S$ is empty, or none of the packages in $S$ are installable. If this is not the case we build three sets. $B_{i+1}$ and $C_{i+1}$ respectively hold the set of all binary packages built so far, and the set of all source packages com-

piled so far. $NS$ is the set of source packages that are left to be built, that is all source packages that are needed to build the binary packages in $NB$ minus the source packages already compiled. The procedure is repeated until all binary packages can be built from the list of source packages and all source packages can be built using the binary packages in the repository.

**Algorithm 4** Build Closure

```
1: procedure F(B_i, C_i, R, S)
2:     NB ← ⋃_{s∈S} IS(s)
3:     if NB = ∅ then
4:         return (B_i, C_i)
5:     else
6:         B_{i+1} ← B_i ∪ NB
7:         C_{i+1} ← C_i ∪ S
8:         NS ← Src(NB) \ C_{i+1}
9:         return F(B_{i+1}, C_{i+1}, R, NS)
10: Closure ←F(∅, ∅, R, S_1)
```

# 4. EXPERIMENTAL VALIDATION

We validate our results using the Debian Sid distribution as of January 2013. To carry out our experiments, instead of using the entire package repository, we selected a self-contained repository by using the build closure algorithm presented in Section 4. The algorithm selected a repository of 613 source packages and 2044 binary packages. This is a representative subset of the full distribution as it contains the base system as well as a number of browsers, window managers, display toolkits and several programming languages like Java, Python and Perl. Because of the structure of the dependency structure of the Debian Sid distribution, considering only a fraction of the packages for our experiments does not change the overall complexity of the problem, but it significantly reduces the background noise produced by thousand of isolated packages and packages with *trivial* build dependencies.

Assuming the existence of a minimal build system, we analyse the dependency graph to compile all source packages, including those that were part of the minimal build system and were thus cross compiled earlier.

There exist fewer dependency cycles during cross compilation than during native compilation. The bootstrapping problem would therefore be easier if more source packages were cross compiled. In our evaluation we chose to cross compile as few source packages as possible (only the minimal build system) because binary packages in most distributions only support native compilation and adding cross compilation support is harder than breaking dependency cycles. Since distribution architects are free to make the minimal build system as big as possible (even including the whole distribution) our tools are also perfectly fit for evaluating the dependency graph in case more or even all source packages were chosen to be cross compiled.

The build graph created from the selected repository contains only two strongly connected components with 977 and 2 vertices respectively. The larger component contains 36 trivial cycles of length two and millions of larger cycles. Using the heuristic presented in the previous sections we were able to remove all cycles by selecting 58 build dependencies to be removed. The total runtime of our algorithms on a standard desktop machine is less then two minutes.

In order to validate the effectiveness of our heuristics, we manually collected and identified a list of optional build dependencies from different sources. We used manually supplied data from package maintainers and automatically harvested information from the Gentoo Linux distribution. Using this data we were able to verify that 88% of the selected build dependencies can be dropped in practice. Moreover it is important to notice that there are two orders of magnitude more build dependencies that could potentially be dropped. By selecting only a few dozens of build dependencies, our algorithms allow to break all dependency cycles with a close to minimal amount of modifications to existing source packages.

Amongst the 88% of removable build dependencies we identify different classes. For example the source package `src:curl` can be compiled without `openssh-server` if a unit test is disabled. The source package `src:gnutls26` can be built without `gtk-doc-tools` if one disables documentation generation. Packages like `src:libxslt` can drop all their dependencies on Python if they do not build their python bindings. The same holds for build dependencies on other languages like Perl, Ruby or Java. Lastly, some source packages can be built with some of their features removed. For example the source package `src:nautilus` provides a configuration option to disable a component which removes its dependency on `libtracker-sparql-0.14-dev`.

It is important to notice that many of the 36 cycles of length 2 can not trivially be broken. Some of those cycles are languages like Python or Vala which need themselves to be built. To bootstrap them, the build system has to be modified to first compile a subset of the whole language which is then used to compile the rest of the language. When, in some circumstances, a cycle of length two can not be broken because the associated build dependencies are not optional, the solution is to cross compile the missing build dependencies.

There is a high correlation between the build dependencies selected by our heuristics with the build dependencies which can be dropped in practice. Our cycle based heuristic selects edges with the most cycles through them. These edges are usually between the highest connected vertices in the source graph. A high degree of connectivity is property of big software package with lots of dependencies or a software package depended upon by many (directly or indirectly). But dependencies on big software packages are mostly optional because the core dependencies of most source packages are small libraries which are easily compilable. Dependencies on big software packages are to generate documentation, run test cases, generate language bindings or generate a functionality specific to that other software package. All of those are usually optional.

If after manual inspection, some of the remaining 12% of selected build dependencies are found to be essential. This additional (negative) information can be fed back into our algorithm to refine our heuristics and produce alternate suggestions to the developer.

### *Build order.*

A topological sorting of the vertices of the source graph results in a linear ordering. According to this order, a bootstrap can be done by compiling one source package after the

other. As many source packages in this linear order do not depend upon each other, they can be compiled in parallel. The partial sorting algorithm we use exploits this fact by selecting packages that are not connected by a $\sim^*$ (Definition 2.12) relationship and marking them for parallel compilation.

The resulting build order consists of 63 groups where all source packages within each group all can be compiled in parallel. The amount of source packages which can be compiled at each iteration quickly drops since packages compiled early have the least amount of build dependencies. Source packages listed last in the build order have the highest build dependency requirements. A final recompilation step ensures that all source packages are compiled with all features enabled and all their build dependencies present.

# 5. EXECUTION PIPELINE

The tools to facilitate build graph analysis and build order creation are designed after the UNIX philosophy. Each tool executes one algorithm, the exchange format between the tools is based on ASCII plain text files and every tool is a filter. Different tasks are carried out by connecting the tools together differently. Execution pipelines for the cross and native bootstrapping phase and to generate self-contained package selections are supplied by shell scripts.

## 5.1 Toolset

The bootstrap toolkit is mainly written in the OCaml programming language and uses dose3 as foundation library [4]. The tool `grep-dctrl` is used for a selection of binary packages which should be available in the final system. The tools `coinstall` and `distcheck` are part of dose3: the former generates a co-installation set while the latter checks installability of binary packages. The `bin2src` and `src2bin` utilities turn a list of binary packages into the list of source packages they build from and a list of source packages into the list of binary packages they build, respectively. The `build_closure` and `build_fixpoint` tools execute the algorithms of the same names, respectively.

## 5.2 Native Pipeline

Figure 7 shows the pipeline for the native phase. Since the cross phase cannot yet be analyzed due to missing metadata its pipeline is omitted. Solid arrows represent a flow of binary packages. Dotted arrows represent a flow of source packages. Dashed arrows represent textual user input. Rectangular boxes represent filters. There is only one input to the filter, which is the arrow connected to the top of the box. Outgoing arrows from the bottom represent the filtered input. Ingoing arrows to either side are arguments to the filter and control how the filter behaves depending on the algorithm. Oval shapes represent a set of packages. Boxes with rounded corners represent set operations.

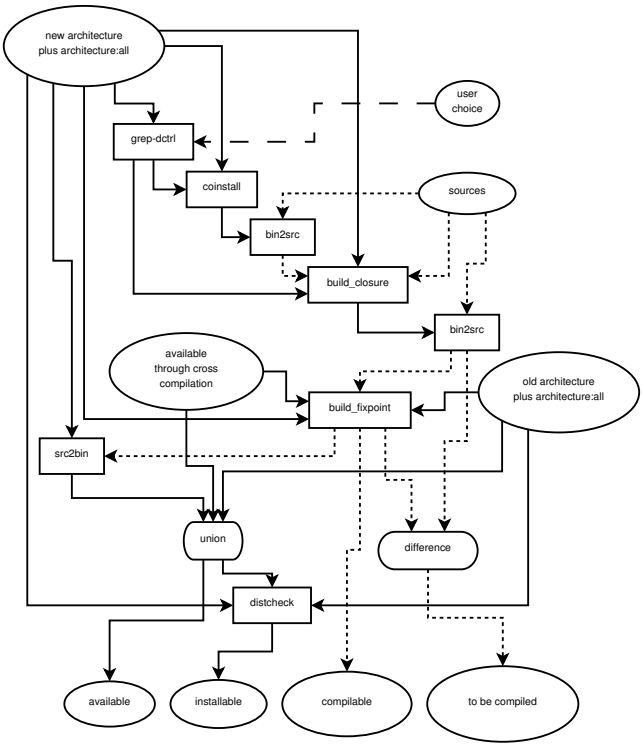



**Figure 7: Native execution pipeline**

# 6. RELATED WORKS

The scope of this paper crosses multiple domains of software engineering, from compilers, to dependency visualization. To the best of our knowledge this is the first work trying to analyse the inter-dependencies derived from the compilation of source packages.

This paper is related to the work of some of the authors, but different in scope: while in [3] we analysed the evolution of a binary repository and how future upgrades of certain packages can affect the entire system, in this paper we analyse the relationship between binary packages and source packages and how the dependency graph can be leveraged to provide useful information to developers to build a repository of binary packages particularly tailored for a specific hardware architecture.

In the literature we can find ideas that are similar to what we have used in this work. In [14], the authors propose the concept of "shared dependencies", that is edges that are common among multiple *simple* cycles (cycles where each vertex appears only once). This is indeed similar to our intuition to select edges in the build graph that are common to multiple cycles. In this work they propose a new layout algorithm to visualize dependencies cycles. However, we doubt that approach can be used in our context where the scale of the problem is at least one order of magnitude larger.

Research of dependency graphs and how to solve cyclic dependencies between software components has so far mainly been carried out for C++ and Java classes or limited to the packaging model of a programming language. PASTA is an interactive refactoring tool that arranges Java classes into hierarchies. The algorithm used involves the removal of cyclic dependency using heuristics that are similar to the ones we present in this paper [10]. Similarly Jooj is an eclipse plug-in used to detect and avoid cyclic dependencies during development using a variant of the feedback arc algorithm [17, 16]. In [5], the authors present a heuristic for automatically optimizing inter-package connectivity and removing dependency cycles based on simulated annealing. To this end, their algorithm picks software classes to be moved between packages.

In the context of compilers and programming languages, the bootstrapping problem has been well studied. A. Appel provides an axiomatization of the bootstrap problem in the context of compilers [6]. In [7], Chambers et all discuss the pitfalls related to software recompilation. While these works are related to the idea of building binary components from source components and dealing with dependency cycles the scale and focus is different from FOSS distributions making difficult a direct reuse of their methodologies.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper we presented a framework to analyse the bootstrap problem. We provided a set of heuristics to aid distribution architects with the daunting task of porting thousands of software packages to a new architecture in a semi-automatic way. The task of finding droppable build dependencies can never be fully automatic as it requires to actively change a software's source code. But once enough source packages are annotated with droppable build dependencies, the task of bootstrapping a distribution will be reduced from a year long manual effort to just the time it takes to automatically recompile every single source package by a build order that can be automatically computed in a matter of minutes. Our experience on the Debian distribution, one of the largest FOSS distributions available, provides a good experimental validation workbench for our hypothesis. As dose3 also supports rpm based distributions, adoption of our tools to support those should not pose significant problems.

We have yet to test the algorithms in a real world bootstrapping setup. The collaboration with the Debian community has been extremely fruitful but still work has to be done to develop an automated bootstrapping tool. For the future we also plan to analyse other FOSS distributions and to extend our approach to other component repositories.

We also want to evaluate other heuristics for example using the concept of strong bridges and strong articulation points in the build graph [11]. Experimental data suggests that strong bridges exist. Their removal would allow to significantly reduce the problem size. This is especially important during the cross phase because the amount of cross compiled packages has to be kept as minimal as possible. We will further investigate if vertex properties like its centrality or shortest distance to a given vertex cluster can be used as helpful heuristics to the developer.

### *Availability.*

All tools developed for this work are available as free and open source software and can be downloaded from the repository `http://gitorious.org/debian-bootstrap/bootstrap`.

The results can be validated by running the `native.sh` shell script in the root of the project's source code repository.


### *Acknowledgements.*

The authors are very grateful to many people for interesting discussions: all the members of the Mancoosi team at University Paris Diderot, Wookey from linaro.org and the Debian community.


# 8. REFERENCES


[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *International Symposium on Empirical Sofware Engineering and Measurement*, pages 89–99. IEEE, 2009.

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

[3] P. Abate, R. D. Cosmo, R. Treinen, S. Zacchiroli, and S. Zacchiroli. Learning from the future of component repositories. In *CBSE*, pages 51–60, 2012.

[4] P. Abate, R. Treinen, R. D. Cosmo, S. Zacchiroli, J. Boender, and J. Zwolakowski. dose3. `http://mancoosi.org/software/#index2h1`.

[5] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui. Automatic package coupling and cycle minimization. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 103–112. IEEE, 2009.

[6] A. W. Appel. Axiomatic bootstrapping: A guide for compiler hackers. *ACM Trans. Program. Lang. Syst.*, 16(6):1699–1718, 1994.

[7] C. Chambers, J. Dean, and D. Grove. Frameworks for intra-and interprocedural dataflow analysis. *Citeseer, Washington University, technical report*, 1996.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2002.

[9] R. D. Cosmo and S. Zacchiroli. Feature diagrams as package dependencies. In *SPLC*, pages 476–480, 2010.

[10] E. Hautus. Improving java software through package structure analysis. In *The 6th IASTED International Conference Software Engineering and Applications*, 2002.

[11] G. F. Italiano, L. Laura, and F. Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447(0):74 – 84, 2012. Combinational Algorithms and Applications (COCOA 2010).

[12] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[13] R. M. Karp. Reducibility among combinatorial problems. *50 Years of Integer Programming 1958-2008*, pages 219–241, 2010.

[14] J. Laval, S. Denier, S. Ducasse, et al. Cycles assessment with cycletable. 2011.

[15] 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*, pages 199–208, 2006.

[16] H. Melton and E. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[17] H. Melton and E. Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In G. Dobbie, editor, *Thirtieth Australasian Computer Science Conference (ACSC2007)*, volume 62 of *CRPIT*, pages 87–95, Ballarat Australia, 2007. ACS.