PAPER

Conflict-Based Checking the Integrity of Linux Package Dependencies

Yuqing LAN[†], ***Member***, **Mingxia KUANG**[†a)], ***and*** **Wenbin ZHOU**[†], ***Nonmembers***

SUMMARY A Linux operating system release is composed of a large number of software packages, with complex dependencies. The management of dependency relationship is the foundation of building and maintaining a Linux operating system release, and checking the integrity of the dependencies is the key of the dependency management. The widespread adoption of Linux operating systems in many areas of the information technology society has drawn the attention on the issues regarding how to check the integrity of complexity dependencies of Linux packages and how to manage a huge number of packages in a consistent and effective way. Linux distributions have already provided the tools for managing the tasks of installing, removing and upgrading the packages they were made of. A number of tools have been provided to handle these tasks on the client side. However, there is a lack of tools that could help the distribution editors to maintain the integrity of Linux package dependencies on the server side. In this paper we present a method based on conflict to check the integrity of Linux package dependencies. From the perspective of conflict, this method achieves the goal to check the integrity of package dependencies on the server side by removing the conflict associating with the packages. Our contribution provides an effective and automatic way to support distribution editors in handling those issues. Experiments using this method are very successful in checking the integrity of package dependencies in Linux software distributions.

key words: *Linux package dependency, conflict, software distribution, integrity of dependencies*

1. Introduction

With the widespread adoption of open source software [1] in all areas of society, the integrity of operating systems based on open source software has become the main direction of the Linux operating system development. Open source software is very often developed in a public, collaborative manner. The source code and certain other rights normally reserved for copyright holders are provided under a software license [2] that permits users to study, change, improve and at times also to distribute the software. These make the interoperability more frequently among open source software. With the increasing size of open source software, the dependency relationships among open source software packages become increasingly complex. The integrity of package dependency is the basis of keeping software and systems running reliably and stably. How to handle a large number of complex dependencies among packages has become the main challenges that maintaining the integrated system stability and the distribution of software packages.

As a result of the open source movement, Linux is notonly being gradually understood and used widely by more and more people, but also being popularly used in commercial applications and becoming one of the mainstream operating systems. As a general operating system, Linux with rich functions can support various major hardware and network, and provide complete applications development and runtime environment. With the development of open source technology, Linux operating systems will provide much more functions which are provided in packages. The calls among the modules are reflected in the dependencies among the software packages.

Maintaining the dependency relationships among packages has always been a difficult task [3]. The main difficulty resides in the fact that a Linux operating system has a large number of software packages, with complex dependencies. These dependency relationships could be easily broken when performing standard life cycle management operations on packages in Linux distribution pool (i.e., package adding, removal and upgrading), leading to unusable and corrupted Linux operating systems.

Linux distribution side, also be referred to as distribution pool [4], is the server side software package resource pool to distribute Linux version and patch though the network. It is composed of different Linux versions and patches. Currently, there are a great many Linux distributions around the world, while Debian [5] is a very large distribution in the world. Debian has strict procedures from the applications to the detection of security patches. It is an operating environment combined with free software, and integrates with a lot of open source software packages. Recently, the number of packages in Debian has sharply increased, but managing the software packages in the distribution side is usually manually done by distribution editors. So, the number of distribution editors is also extremely increased. In fact, Debian has already included almost all packages (nearly more than 30000), which requires a lot of distribution editors to manage the software packages. The same problems appear in domestic Linux related versions, such as NeoShine Linux. The number of NeoShine Linux server 5.0 packages reaches 2270, and this number is continuously increasing with the expansion of functions.

To ensure the integrity of Linux package dependencies of the distribution pool is the basis of distributing and updating continuously through the network. Since the update of packages persistent changes in the distribution pool, for a dependency integrity distribution pool, (each package in the distribution pool can be installed), integrity maintenance re-

Manuscript received May 9, 2011.
Manuscript revised August 15, 2011.
[†]The authors are with Beihang University, 100191, China.
a) E-mail: kuangmingxia@cse.buaa.edu.cn
DOI: 10.1587/transinf.E94.D.2431

quires distribution editors to check the installability of all the new packages joined to the distribution pool, and check the dependency integrity if remove some packages from the distribution pool. Since a distribution version is composed of thousands of packages, the scale of Linux distribution side become larger and the manual maintenance costs become higher. To reduce maintenance costs, we are facing to demand of automated management of packages in the Linux distribution side, so as to reduce maintenance costs.

The current tools for the management of complex dependencies among packages can be divided into two types from the use aspect: client-based package dependency management tools and distribution-based package dependency management tools. The client-based package dependency tools can install and upgrade packages with all its dependencies automatically downloaded from software warehouse through internet. The distribution-based package dependency management tools are used to maintaining the integrity dependency of the distribution pool. Linux distribution versions have provided package dependency management tools for the client side, such as Debian Linux' apt-get [4], RedHat Linux's yum [6], Suse Linux' red carpet [7], etc. Several researchers developed and optimized better packages management tools (i.e. Smart [8], Optimal [9]) aimed at the incompleteness problems of apt-get and yum. On the distribution side, the famous Linux operating system vendors such as RedHat, Debian and Suse have carried out the relevant work in the field of package dependencies, and have proposed methods of solving the problems of package dependencies. Red Hat Enterprise focuses on solving the package dependencies based on RPM [10] package format and Debian Linux uses DEB [11] package format to achieve this purpose. However the research result is the enterprise inner secrets and the design and implementation cannot be acquired generally. Domestic (China) manufacturers have not developed an effective way to solve the package dependency problems, and all these tasks are done manually.

Several researchers use the SAT method to solve the problem of package dependencies. SAT problem is a basic NP-complete problem. In recent years, SAT algorithm has already made a number of representative results through sustained and in-depth researched [12]–[14]. Paper [15] presented a formal description of package dependencies and introduced a set of mapping rules that transform dependency problem into Boolean SAT problem. At last, it is designed a basic algorithm based on the art-of the state SAT solver MiniSAT. EDOS European project [16] carried out researches on the decision method of the package dependencies in Linux distribution side. A law was proposed by the project: Given a package repository R and the package p, the installable problem of p is an NP-complete problem. Therefore, through mapping the package dependencies to SAT problems [17]–[19] and using the SAT mature method, we can get a satiable solution of package dependencies. The main idea of this method is to find a solution that makes all the package dependency satiable. However, the EDOS' source cannot check the dependency in the distribution side.

The widespread adoption of Linux operating systems, in some way, has emphasized these problems. How to effectively maintain the relationships among abundant and complex packages has become the challenge that all Linux vendors need to face. Does the new package conflict with other package or not? Do all dependencies of the new package exist in the package set or not? Does the original dependency relationship exist or not, when the package is updated? Dose the original dependency relationship still maintain or not, when someone package is deleted or abandoned? How to quickly develop a dedicated Linux operating system release which contains all necessary packages? The resolution of these issues in the final analysis is focused on resolution of package dependency.

In this paper we attach a great important package dependencies research and propose a new method based on conflict to check the integrity of Linux package dependencies. The method excludes the conflict associated with a package to check the integrity of the Linux server distribute version from the perspective of conflict. The results of our research can help Linux vendors to maintain large package bases and improve the quality of software distributions built on them, by detecting errors and inconsistencies in an effective, fast and automatic way. Due to the popularity of RPM and DEB packages in industry, the checking method supports both of them.

This paper is structured as follows: in Sect. 2, we address the related theory. Section 3 presents the algorithm that we have used to reason on our method, and Sect. 4 shows the results we have gathered from the analysis of some NeoShine Linux server versions. Finally, in Sect. 5 we draw conclusions.

2. Related Theory

2.1 Package Formats Analysis

This section briefly describes some package formats. Package metadata file is the file which describes the package information including the name, version, size, dependencies and conflicts and so on. This paper mainly presents the method extracting dependency information from package metadata file and uses the dependency information to check the integrity of the package dependencies. We focus on package-based GNU/Linux distributions: the DEB and the RPM formats. RPM package format is used in Red Hat, Fedora, SUSE and domestic (China) manufacturers while DEB is adopted in Debian and Ubuntu. Both of these two formats are used in current major Linux.

Though DEB and RPM packages are different, they have a lot of commonalities. In what follows, we describe the features that are relevant to the topic of this paper: dependency specification.

DEB has two types: binary packages and source packages. Binary packages can be directly installed while the source code included in the source packages can be used to create binary packages. A source package can create mul-

tiple binary packages. Each DEB package, binary package and source package, includes package's version number, control files and installation source files. The composition of a basis DEB package ocaml 3.08.3-8 i386.deb can be shown as follows:

ocaml 3.08.3-8 i386.deb

- debian-binary (version)
- control.tar.gz
  - ostinst (post-install script)
  - prerm (pre-removal script)
  - postrm (post-removal script)
  - md5sums (MD5 sums for data.tar.gz)
  - control (package metadata)
- data.tar.gz
  - /usr
  - /usr/lib
  - /usr/lib/ocaml/3.08.3

From its composition, we can find that DEB package is an actually archive including package's version and two compressed files. One of the compressed file contains the description control information while the other is the installation file.

RPM is an archive too and is actually an ad-hoc format explicitly conceived for this purpose. DEB packages can be produced using standard tools such as ar and tar, so they can be easily managed. Nevertheless, the most relevant difference between DEB and RPM package format concerns their metadata specification. While RPM packages encode it in a binary form as a part of its ad-hoc archive format, DEB packages use a textual representation for it, which makes its processing easier.

Each package uses the unique version as an important sign. The version number is a key factor to identify the relationship among the packages. The most of dependency categories described in the software package metadata file are the same except some special dependency relationship such as obsoletes and replaces. But these dependency relationships almost have no impact on the integrity of the packages. Therefore, this paper extracts three dependencies for the basis of checking the integrity of the packages [20].

Depends (DEB), Requires (RPM): It is used to express the dependency packages of present package. If the package can be installed or run normally, these packages must be in Linux operating systems.

Conflicts (DEB, RPM): It is used to express the conflict packages of present package. If the current package can be normally installed or run, the conflict packages cannot be in the Linux operating system at the same time.

Pre-Depends (DEB), PreReq (RPM): It is similar to Depends relationships but it is used to establish a requirement on the packages that must be already presented in the Linux operating system in order to successfully deploy the packaged component. The difference between Pre-Depend and Depends is that while Depends package might not be presented in the Linux operating system when deploying the packaged component (but only after, so they can be deployed together with the current component), while Pre-Depends packages must be already installed even before attempting to deploy the current packaged component.

The dependency relationships of a package are specified by using a list of package names, optionally with version constraints. Each element of the list represents a relationship. When every element of the list is satisfied, the dependency relationship is satisfied. Actually, an element of the list doesn't contain only one package. In the DEB format package, the element allows several packages with disjunctive relationship. This is done by using the (" |") operator [21]. In this situation, in order to meet the disjunctive relationship it is sufficient that at least one of the constituting dependency relationships is met. In other words, one of these packages is satisfied, the element dependency is satisfied. The element like this is called composite element. Take the package mysql for example, package name: mysql-server-5.0, version: 5.0.51a-3ubuntu5.4, the Depends contains the dependency libc6(>= 2.4), this represents package mysql depends on package libc6 and the version of libc must greater than or equal to 2.4. If there are two versions of libc6, 2.4 and 2.6, in a Linux operating system, the Depends is described as libc6(2.4) or libc6(2.6).

2.2 Relationship Among Packages

In general, the relationship between any two packages can be divided into three categories: dependency relationship, conflict relationship and no relationship. Dependency relationship is a very common relationship. To install package p_1 we must first install the package p_2 which means there has a dependency relationship between p_1 and p_2. Dependency relationship can be divided into direct dependency and indirect dependency. For example, package a provides the function P which has to use the function Q provided by package b. We say that the package a depends on the package b. If the package b depends on package c then the package a also depends on c. The relationship between package a and b is called direct dependency and the relationship between package a and c is called indirect dependency. The dependencies of package a, b and c form a dependency chain. If b or c is lost, the dependency chain is destroyed. Conflict relationship is rare among actual packages, and it is a negative dependency, that is, package p_1 and p_2 could not co-exist.

In software package repository, if the dependencies of every package are existent and there is no conflict between any two packages, it is called the dependencies of software package are satisfied, otherwise it is called the dependencies are not satisfied. In order to ensure stable operation and normal function of the Linux operating system, we must firstly ensure the package set repository constituting the operating system version is dependencies satisfied. It is important for Linux operating system to be commercial applications.

Inter-package dependencies can be described as a directed graph [22], the relationships among nodes can be

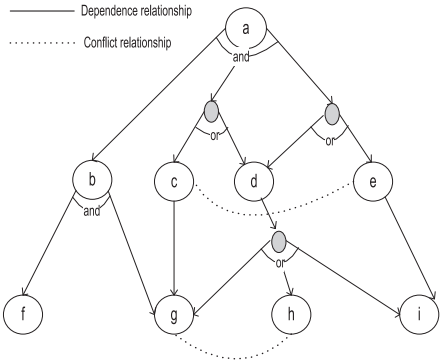

Fig. 1 The relationships composed of nice packages.

divided into dependence relationship and conflict relationship. The "and dependence" and "or dependence" can be expressed by the extended "and-or" graph [23]. The dotted line indicates the conflict relationship and solid line denotes the dependence relationship.

For example, in a Linux distribution, the relationship of package a is shown in Fig. 1. From this figure, we can find that the total number of the direct dependency and indirect dependency of package a is eight. The direct dependencies of package a are b, c or d, d or e. The direct dependencies of package b are f and g and so on.

If we want to install package a, we have to install package b, c or d, d or e. Package c and e cannot co-exist in the same system for the conflict relationship.

3. Algorithm Design

3.1 Basic Definition

This section will present some definitions to be use in the following sub-sections and the notions of checking the integrity of package dependencies.

Definition 1 (Unit package can be installed): The unit package is a set of files containing metadata, program files, installation scripts and configuration information. It is the basis unit composed by software and it is called software package in the Linux operating system. We focus on the management of dependency relationship among unit packages in the Linux operating system, therefore, the *unit package can be installed* will be called package in the follow. A package is a pair (u, v) where u is an identifier and v is a version, marked as $p(u, v)$. In addition, $u(p)$ is the identifier of package p and $v(p)$ is the version of package p. We introduce the definition of unit package can be installed that identify the central property we want to gurantee for each package present in a package repository.

Definition 2 (Package repository): A package repository is a tuple $R = (P, D, C)$ where P is a set of packages, $D : P \to \psi(\psi(P))$ is the dependency function (we write $\psi(X)$ for the set of subsets of X), and $C \subseteq P \times P$ is the conflict relationship. The relationship C is symmetric, i.e., $(p_1, p_2) \in C$ if and only if $(p_2, p_1) \in C$ for all $(p_1, p_2) \in P$.

As for a Linux operating system distributions version, two packages with the same identifier but different versions conflict, that is, if $p_1 = (u, v_1)$ and $p_2 = (u, v_2)$ with $v_1 \neq v_2$, then $(p_1, p_2) \in C$.

In a package repository R, the dependencies of each package p are given by $D(p) = \{d_1, \cdots, d_k\}$ which is a set of sets of packages, interpreted as follows. If p is to be installed, then all its k dependency requirements must be satisfied. For d_i to be satisfied, at least one of the packages in d_i must be available. In particular, if one of the d_i is an empty set, it will never be satisfied, and the package p is not installable. If every package in the package repository R is dependency satisfied, then all the packages can be installed, which means the package repository R dependency is integrated. The language to express package relationships (including dependence relationship and conflict relationship) is not as simple as flat lists of component predicates, but rather a structured language whose syntax and semantics is expressed by conjunctive normal form (CNF) formulae [20].

For example, Fig. 1 represents the relationships among the packages in a package repository R.

The relationships among packages can be formalized as follows:

$$\begin{aligned} &P = \{a, b, c, d, e, f, g, h, i\} \\ &D(a) = \{\{b\}, \{c, d\}, \{d, e\}\} \\ &D(b) = \{\{f\}, \{g\}\}, D(c) = \{\{g\}\} \\ &D(d) = \{\{g, h, i\}\}, D(e) = \{\{i\}\} \\ &C = \{(c, e), (e, c), (g, h), (h, g)\} \end{aligned} \tag{1}$$

Where package a is the target and if you want to install a, the following packages must be installed: b either c or d, either d or e. Package c and e, g and h cannot be installed at the same time. Thus the general form for package a dependency specification is a conjunction of disjunctions:

$$a \to (b_1^1 \vee \cdots \vee b_1^{r_1}) \wedge \cdots \wedge (b_n^1 \vee \cdots \vee b_n^{r_n}) \tag{2}$$

For package a to be installed, each term of the right-hand side of the above implication must be satisfied. In turn, the term $b_i^1 \vee \cdots \vee b_i^{r_i}$ when $1 \leq i \leq n$ is satisfied when at least one of the b_i^j with $1 \leq j \leq r_i$ is satisfied. If a is a package in our package repository R, we therefore have

$$D(a) = \{\{b_1^1, \cdots, b_1^{r_1}\}, \cdots, \{b_n^1, \cdots, b_n^{r_n}\}\} \tag{3}$$

Where, $D(a)$is the direct dependency collection of the package a.

In particular, if one of the terms is empty (if $\phi \in D(a)$), then package a cannot be satisfied.

Definition 3 (Packages can be installed): For a software package repository R and a package p, if all the dependence relationship of the package p is satisfied in the package repository R, the package p can be installed in the package repository R.

Definition 4 (Dependency Collection): The dependency collection of the package a refers to the set composed of all the sets of packages which the package a depends

on. We write $D_r(a)$ for the dependency collection of the package a, if $a \rightarrow (b_1^1 \vee \cdots \vee b_1^{r_1}) \wedge \cdots \wedge (b_n^1 \vee \cdots \vee b_n^{r_n})$ then $\{b_i^1, \cdots, b_i^{r_i}\} \in D_r(a)$ where $1 \leq i \leq n$, and if $b_i^j \rightarrow (c_1^1 \vee \cdots \vee c_1^{s_1}) \wedge \cdots \wedge (c_m^1 \vee \cdots \vee c_m^{s_m})$ with $1 \leq j \leq r_i$, then $\{c_i^1, \cdots, c_i^{s_i}\} \in D_r(a)$, where $1 \leq i \leq m$.

For example, Fig. 1 shows the relationship composed of nine packages. The dependency collection of the package a is $D_r(a) = \{\{b\}, \{c, d\}, \{d, e\}, \{f\}, \{g\}, \{g, h, i\}, \{i\}\}$.

Definition 5 (Conflict Collection): We write $D_c(a)$ for the conflict collection of the package a. The conflict collection $D_c(a)$ refers to the set composed of all the sets of packages which conflict with anyone in the dependency collection $D_r(a)$. We can formalized as follows: if b_i is a package, $b_i \in \{b_1, \cdots, b_n\}$ and $\{b_1, \cdots, b_n\} \in D_r(a)$, when package b_i conflicts with package c, then $\{c\} \in D_c(a)$.

For example, Fig. 1 shows the relationship composed of nine packages. The conflict collection of package a is $D_c(a) = \{\{c\}, \{e\}, \{g\}, \{h\}\}$.

Definition 6 (Dependency Priority): In the dependency collection $D_r(a)$ of the package a, if $D(a) = \{\{b_1^1, \cdots, b_1^{r_1}\}, \cdots, \{b_n^1, \cdots, b_n^{r_n}\}\}$ then the dependency priority of set $\{b_i^1, \cdots, b_i^{r_i}\}$ is 0 (the superlative), where $1 \leq i \leq n$. And if we have the set $\{b_i^1, \cdots, b_i^{r_i}\}$ with $r_i = 1$ (which means the set has only one element), and $D(b_i^j) = \{\{c_1^1, \cdots, c_1^{s_1}\}, \cdots, \{c_m^1, \cdots, c_m^{s_m}\}\}$ with $1 \leq j \leq r_i$, then the dependency priority of term $\{c_t^1, \cdots, c_t^{s_t}\}$ is 0, where $1 \leq t \leq m$. If we have the set $\{b_i^1, \cdots, b_i^{r_i}\}$ with $r_i > 1$ (which means the set has more than one element), the dependency priority of $\{c_t^1, \cdots, c_t^{s_t}\}$ is lower than $\{b_i^1, \cdots, b_i^{r_i}\}$, which is 1, where $1 \leq t \leq m$. If a package set has many dependency priority's values, we only keep the highest dependency priority.

For example, Fig. 1 shows the relationship composed of nine packages. $D(a) = \{\{b\}, \{c, d\}, \{d, e\}\}$, so the dependency priority of $\{b\}$ is 0, the same with $\{c, d\}$, $\{d, e\}$. $D(b) = \{\{f\}, \{g\}\}$, so the dependency priorities of $\{f\}$ and $\{g\}$ are 0. $D(c) = \{\{g\}\}$, so the dependency priority of package $\{g\}$ is lower than $\{c, d\}$, which is 1. From the Fig. 1, we can see that $D(b) = \{\{f\}, \{g\}\}$ and $D(c) = \{\{g\}\}$, so the dependency priority of $\{g\}$ has two values: 0 and 1. According to the definition of Dependency Priority, we keep the higest one, so the dependency priority of $\{g\}$ is 0. $D(d) = \{\{g, h, i\}\}$, so the dependency priorities of $\{g, h, i\}$ is lower than $\{c, d\}$ or $\{d, e\}$, which is 1. $D(e) = \{\{i\}\}$, so the dependency priority of $\{i\}$ is lower than $\{d, e\}$, which is 1.

Definition 7 (Conflict Priority): In the conflict collection $D_c(a)$ of the package a, if $\{c\} \in D_c(a)$ and the package c conflicts with package b_i where $b_i \in \{b_1, \cdots, b_n\}$ and $\{b_1, \cdots, b_n\} \in D_r(a)$ when $n > 1$ (which means the set $\{b_1, \cdots, b_n\}$ has more than one element), the conflict priority of $\{c\}$ is lower than the dependency priority of $\{b_1, \cdots, b_n\}$. If $n = 1$, then conflict priority of $\{c\}$ is the same as dependency priority of package $\{b_1, \cdots, b_n\}$. If a package set has many conflict priority's values, we only keep the highest conflict priority.

Following the above example, Fig. 1 shows package c and e, g and h cannot be installed at the same time. Package c conflicts with e in the $\{d, e\}$, so the conflict priority of $\{c\}$ is lower than the dependency priority of $\{d, e\}$, which is 1. Package e conflicts with c in the $\{c, d\}$, so the conflict priority of $\{e\}$ is lower than the dependency priority of $\{c, d\}$, which is 1. Package g conflicts with h in the $\{g, h, i\}$, so the conflict priority of $\{g\}$ is lower than the dependency priority of $\{g, h, i\}$, which is 2. Package h conflicts with the g in the $\{g\}$ and the g in the $\{g, h, i\}$, we keep the highest priority value, so the conflict priority of $\{h\}$ is the same with dependency priority of $\{g\}$, which is 0.

The Dependency Priority is used to help us calculate the Conflict Priority. And we introduce the definition of Conflict Priority in order to provide a processing sequence of packages in the Conflict Collection.

3.2 Mathematical Description and Theoretical Analysis

Formalization of package direct dependency relationship: as the definitions of Sect. 3.1, we write D_r and D_c for the dependency and conflict collections respectively, and D for the direct dependency collections. If there is no dependency relationship between packages p and any other packages in the repository R, then $D_r(p) = \phi$. If no conflict relationship, then $D_c(p) = \phi$. Assignment λ on the direct dependency collection D defined as $D^\lambda(p)$, is making one assignment of the disjunction in the collection, which means you only need to select one package in each disjunction. After assignment, $D^\lambda(p)$ is one of direct dependency collections of p.

The dependency satisfaction is also known as the integrity of dependencies. For a software package repository R, the dependency satisfaction of package p in the repository R is an assignment λ, where $D^\lambda(x)$ is the dependency function and expresses the direct dependency collection of the package x. $D^{n\lambda}(p)$ denotes $D^\lambda(D^\lambda(D^\lambda(\ldots D^\lambda(p)\ldots)))$, for any one package x of sub-collection $p = \bigcup_{n \geq 1} D^{n\lambda}(p)$, where $p \notin D(x)$, $p \supset D_r^\lambda(x)$ and $p \bigcap D_c^\lambda(x) = \phi$.

If all the dependency packages of a package exist, the reason of leading to unsatisfied dependency is the conflict packages among the dependency collection. The proposed method can solve the problem of the dependencies. The method firstly checks the existence of the direct dependency packages, and then deals with the conflict packages in the conflict collection. If the dependency packages not exist or the conflict packages cannot be resolved, the dependencies of the package are not satisfied. The flow diagram is shown in Fig. 2.

The key of the method is to process conflict relationships. The conflict relationships can be divided into solvable conflict and unsolvable conflict. The solvable conflict is that the conflict target package is not in the dependency collection or is in the dependency collection but the priority is not superlative and can be deleted. The package which can be deleted is not a superlative package. Delete a package meanwhile delete the packages which depend on it. When the compound dependency is only one item, it cannot be deleted; meanwhile the priority which is 0 cannot be

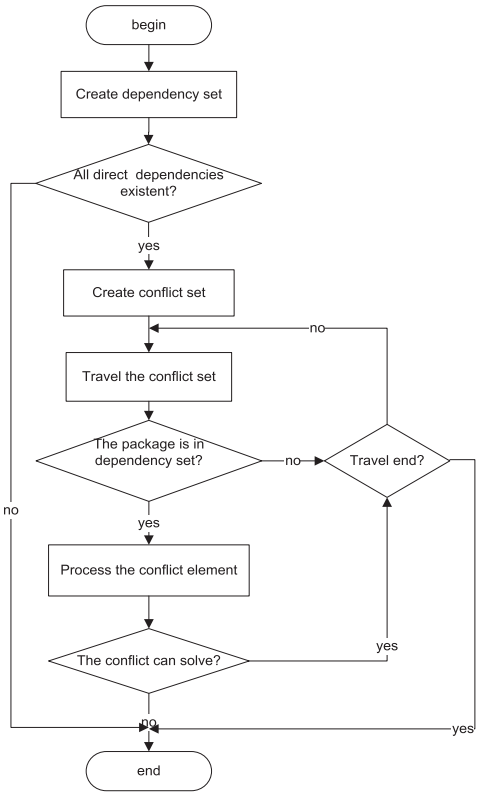

Fig. 2 The flowchart of algorithm.

deleted.

Take Fig. 1 as a example, if we want to install package a, we have to install package b, either c or d, and either d or e. Package c and package e cannot exist in the same system for the conflict relationship. As described in the first part of Sect. 3, we defined the symbol $D_r(a)$ as the dependency collection of the package a, the symbol $D_c(a)$ as conflict collection of the package a and the symbol $D(a)$ as the direct dependency collection of the package a. The number joined by the "-" is used to represent the dependency priority of package. The smaller the number is, the higher the dependency priority becomes. The direct dependency collection and dependency collection of the package a express as follows.

$$\begin{aligned} D(a) &= \{\{b\},\{c,d\},\{d,e\}\} \\ D_r(a) &= \{\{b-0\},\{c,d-0\},\{d,e-0\},\{f-0\}, \\ &\quad \{g-0\},\{g,h,i-1\},\{i-1\}\} \end{aligned} \tag{4}$$

In the direct dependency collection $D(a)$, if any one does not exist, then package a is not installed for losing the dependence package. If all the dependence packages exist, we construct the $D_c(a)$ from the $D_r(a)$ and show as follows.

$$D_c(a) = \{\{h-0\},\{e-1\},\{c-1\},\{g-2\}\} \tag{5}$$

In the conflict collection $D_c(a)$, the number joined by the "-" is used to represent the conflict priority of package. The smaller the number is, the higher the conflict priority becomes. Then sort the element of the $D_c(a)$ by the priority.

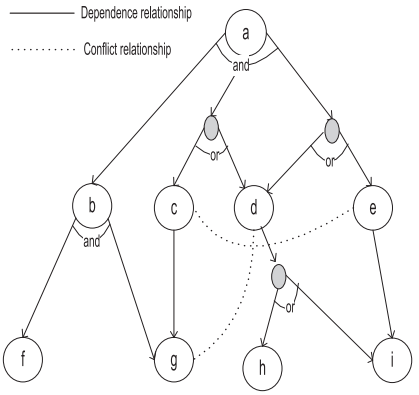

Fig. 3 The dependency relationships of package a.

We will process the conflict packages from high conflict priority to low. For the element $\{h-0\}$, the conflict packages of package h exist in dependency collection $D_r(a)$ and can be processed, so we remove the package h from $D_r(a)$. After processing the element $\{h-0\}$, the result of the $D_r(a)$ and $D_c(a)$ is

$$\begin{aligned} D_r(a) &= \{\{b-0\},\{c,d-0\},\{d,e-0\},\{f-0\}, \\ &\quad \{g-0\},\{g,i-1\},\{i-1\}\} \end{aligned} \tag{6}$$

$$D_c(a) = \{\{e-1\},\{c-1\}\} \tag{7}$$

Continue to process the $D_c(a)$ until the element of $D_c(a)$ is null. After processing the element $\{e-1\}$ or $\{c-1\}$, the result of $D_r(a)$ and $D_c(a)$ is:

$$\begin{aligned} D_r(a) &= \{\{b-0\},\{c,d-0\},\{d-0\},\{f-0\}, \\ &\quad \{g-0\},\{g,i-1\}\} \end{aligned} \tag{8}$$

$$D_c(a) = \{\}$$

Or

$$\begin{aligned} D_r(a) &= \{\{b-0\},\{d-0\},\{d,e-0\},\{f-0\}, \\ &\quad \{g-0\},\{g,i-1\},\{i-1\}\} \end{aligned} \tag{9}$$

$$D_c(a) = \{\}$$

From the result of $D_r(a)$ and $D_c(a)$, we can know the dependency of the package a is satisfied. A optimum solution is $\{b,d,f,g\}$.

As shown in Fig. 3, suppose package g conflicts with package d rather than conflict with h, the result of $D_r(a)$ and $D_c(a)$ is as follows:

$$\begin{aligned} D(a) &= \{\{b\},\{c,d\},\{d,e\}\} \\ D_r(a) &= \{\{b-0\},\{c,d-0\},\{d,e-0\}, \\ &\quad \{f-0\},\{g-0\},\{h,i-1\},\{i-1\}\} \end{aligned} \tag{10}$$

$$D_c(a) = \{\{d-0\},\{e-1\},\{c-1\},\{g-1\}\} \tag{11}$$

Then process the element of the $D_c(a)$ from highest priority to lowest priority. For the element $\{d-0\}$, the conflict packages exist in the dependence collection $D_r(a)$ and can be processed, so we remove the element $\{d-0\}$. After processing element "$d-0$", the result of $D_r(a)$ and $D_c(a)$ as

follows:

$$D_r(a) = \{\{b-0\},\{c-0\},\{e-0\},\{f-0\},\{g-0\},\{i-1\}\} \quad (12)$$

$$D_c(a) = \{\{e-1\},\{c-1\}\} \quad (13)$$

Continue to process element $\{e-1\}$ or $\{c-1\}$ and the result of $D_r(a)$ and $D_c(a)$ is as follows:

$$D_r(a) = \{\{b-0\},\{c-0\},\{\Phi-0\},\{f-0\},\{g-0\}\} \quad (14)$$

$$D_c(a) = \{\}$$

Or

$$D_r(a) = \{\{b-0\},\{\Phi-0\},\{e-0\},\{f-0\},\{g-0\},\{i-1\}\} \quad (15)$$

$$D_c(a) = \{\}$$

There is a null element Φ in the dependency collection $D_r(a)$, therefore the dependency of the package a cannot be satisfied.

Hence, to check installability of a package p, firstly, get the dependency collection and conflict collection of package p. Then traverse the dependency collection to check the packages that package p depended on are all present. If not, then package p is not installable. Otherwise, process the conflict element of the conflict collection one by one according to the conflict priority. If some package in the conflict collections could not be processed, the dependency of package p is not satisfied.

4. Experiment and Analysis

In order to validate the correctness and feasibility of the checking method, we have developed a tool according to the checking method. It can be used by a distribution editor to manage the integrity of Linux package dependencies. In this section we show some experimental results that we have gathered by analyzing with our tool of NeoShine Linux server version 5.0 and NeoShine Linux server version 5.3. There are eight build version 5.0 and nine build version 5.3. The result of the checking is shown in Table 1.

Based on the data of Table 1, we have drawn the cylindrical statistical diagram Fig. 4 and Fig. 5.

From Fig. 4 and Fig. 5, it is clear to see with the Linux Server build version increase, the number of packages of the lack of dependence and the number of unsatisfied dependency are gradual decrease, especially in the Linux Server 5.3 build06-09, the missing dependency packages is zero.

During the experiment, we found that there exist some conflict packages in the tested Linux versions, but the unsolvable conflict relationships among the packages don't exist in nearly all mature Linux operating systems. As can be seen from Table 1, at least the 17 tested versions didn't appear the unsolvable conflict, i.e., "Number of conflict cannot be solved" is 0. For example, in NeoShine Server 5.0 build-09, the package "gnome-spell" which the package "evolution" depends on didn't exist and in NeoShine Server 5.3 build-03, the package "redhat-lab" on which the package "sblim-cmpi-network-devel" didn't exist and so on.

During checking the integrity of dependency, some

Table 1 The result of the integrity checking of Linux server dependency.

Version	number of packages	number of missing dependency packages	number of dependence unsatisfied	number of conflict packages	number of conflict can not be resolved
NS5.0-b07	2271	11	102	280	0
NS5.0-b08	2270	11	102	281	0
NS5.0-b09	2270	10	54	280	0
NS5.0-b10	2270	10	54	280	0
NS5.0-b11	2270	10	54	281	0
NS5.0-b13	2270	10	54	280	0
NS5.0-b14	2249	9	18	271	0
NS5.0-b16	2249	9	18	271	0
NS5.3-b01	2243	5	151	281	0
NS5.3-b02	2245	4	147	281	0
NS5.3-b03	2245	4	57	281	0
NS5.3-b04	2254	4	57	281	0
NS5.3-b05	2219	1	35	286	0
NS5.3-b06	2234	0	0	288	0
NS5.3-b07	2234	0	0	288	0
NS5.3-b08	2234	0	0	288	0
NS5.3-b09	2234	0	0	288	0

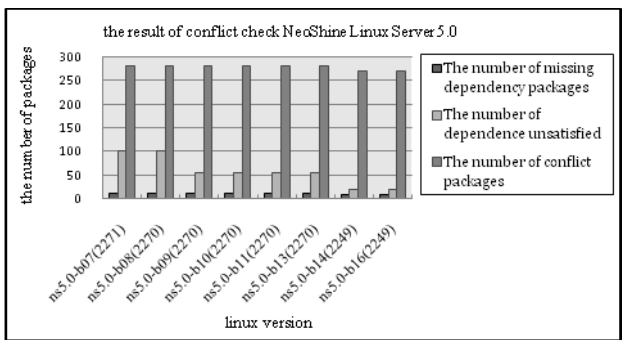

Fig. 4 NeoShine Linux Server 5.0 dependency check result.

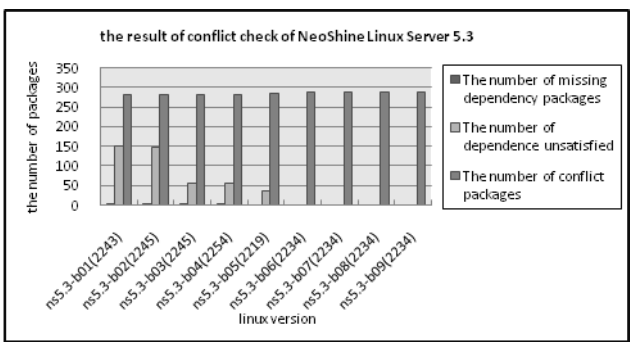

Fig. 5 NeoShine Linux Server 5.3 dependency check result.

conflict problems were found for the wrong metadata description. For example, the package "gtk2" appearing in multiple Linux versions conflict with the "libgnomeui" whose version less than "libgnomeui-2.15.1cvs20060505-2". It is clear that the version description of the package is wrong or in other words it doesn't follow the certain norms. The package metadata information was corrected by communicating with the developers.

The problems in the above process of checking were confirmed by discussing and exchanging with the developers. This fully shows that proposed method in this paper is correct. During the experiment, we discovered that some packages whose dependency is not satisfied can be installed in special sequence and environment.

The given notions of dependency priority, conflict priority and conflict-based checking method can be used to address issues showing up in the maintenance of large number of Linux operating systems packages.

5. Conclusion

The package management has become more and more important for every Linux enterprise. However there is still a lack of related researches on the distribution package management. This paper presents a conflict-based checking method of package dependency integrity, by solving the conflict to find the dependency satisfied solution. The method completes the integrity check of the distribution side by checking the dependency integrity of the packages. This method provides an effective way for the integrity check of the Linux distribution side.

Compared with the currently existing technology, our proposed approach has obvious advantages and useful effects. Conflict relationship, a negative dependency, is rare, which means p_1 and p_2 package could not co-exist. Linux operating system allows each package with one version. Therefore many conflict packages do not co-exist in one system. So from the perspective of the conflict, checking the package dependencies is more efficient in distribution side.

In summary, the proposed approach has the above advantages and beneficial effects. There is a significant progress in technologies and extensive use of industry value.

Acknowledgements

Thank the staff of the China Standard Software CO., LTD for their help. Without their careful guidance and help, we cannot accomplish the experiments so smoothly. Thank all those who helped us.

References

[1] D. Weiss, "Quantitative analysis of open source projects on SourceForge," Proc. First International Conference on Open Source Systems, pp.140–147, Genova, 2005.
[2] Opensource.org. Opensource licenses. http://www.opensource.org/licenses
[3] R.D. Cosmo, P. Trezentos, and S. Zacchiroli, "Package upgrades in FOSS distributions: Details and challenges," HotSWup'08, 2008.
[4] R.D. Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon, "Maintaining large software distribution," New Challenges from the Foss era. 1st International EASST-EU Workshop on Future Research Challenges for Software and Services, Vienna, 2006.
[5] Debian Group. Debian Policy Manual. http://debian.org/doc/debian-policy/, [EB/OL].
[6] Fedora, yum, http://fedoraproject.org/wiki/Tools/yum, accessed Jan. 2010.
[7] derkeiler, Red carpet, http://linux.derkeiler.com/Mailing-Lists/SuSE/2005-09/2179.html, accessed Dec. 2009.
[8] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "OPIUM: Optimal package install/uninstall manager," Proc. 29th International Conference on Software Engineering (ICSE'07), pp.178–188, 2007.
[9] G. Niemeyer, Smart package manager, http://labix.org/smart, 2006 [EB/OL].
[10] E.C. Bailey, Maximum rpm. http://www.rpm.org/max-rpm.
[11] The Debian Project. Debian policy manual. http://www.debian.org/doc/debian-policy/index.html.
[12] N. Een and N. Sorensson, "An extensible SAPsolver. 6m intemational conference," SAT 2003. LNCS 2919: 502. 518.
[13] M. Davis and H, Putnam, "A computing procedure for quantification theory," J. Association for Computing Machinery, vol.7, pp.20l–215, 1960.
[14] J.P. Marques-Silva and K.A. Sakallah, "GRASP-A new search algorithm for satisfiability," Proc. International Conference on Computer Aided Design (ICCAD), 1996.
[15] H. Gu and X.-Z. Ni, "SAT-based analysis of package dependency problem," The Ninth Graduate Symposium on Computer Science and Technology of Chinese Academy of Sciences Institute of Computing Technology, 2006.
[16] EDOS, http://www.edos-project.org/.
[17] J. Marques-Silva, "Practical applications of Boolean satisfiability," Discrete Event Systems, 2008. WODES 2008. 9th International Workshop, pp.74–80, May 2008.
[18] J. Rintanen, K. Heljanko, and I. Niemela, "Planning as satisfiability: Parallel plans and algorithms for plan search," Artif. Intell., vol.170, no.12-13, pp.1031–1080, 2006.
[19] I. Lynce and J. Marques-Silva, "Efficient haplotype inference with Boolean satisfiability," National Conference on Artificial Intelligence, July 2006.
[20] 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," ASE, pp.199–208, 2006.
[21] P. Abate, R.D. Cosmo, J. Boender, and S. Zacchiroli, Empirical Software Engineering and Measurement.
[22] N. LaBelle and E. Wallingford, Inter-package dependency networks in open-source software. CoRR, cs.SE/0411096, 2004.
[23] Y.-Q. Lan, X.-G. Duan, J. Gao, W.-B. Zhou, and H. Zhao, "Extraction methods on Linux package dependency relations," Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference, Dec. 2009.

Yuqing Lan born in May 1969, Ph.D., Associate Professor in the Computer science and engineering school of Beihang University, specialized in software engineering and operating system, published 38 papers, 21 of them are included in EI. This work is partially supported by the National Project of Core Electronic Devices, High-end General Chips and Basic Software-"Domestic basic software testing for integration and application" under Grant No. 2009ZX01045-005-002.

Mingxia Kuang born in Aug. 1986, graduate student, studying at Beihang University, Research Interest is Software Engineering.

Wenbin Zhou born in July 1985, graduate student, studying at Beihang University. Research Interest is Software Engineering.