The Journal of Systems and Software 85 (2012) 2228–2240

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Dependency solving: A separate concern in component evolution management☆

Pietro Abate[a], Roberto Di Cosmo[a,b], Ralf Treinen[a], Stefano Zacchiroli[a,*]

[a] Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France
[b] INRIA Paris-Rocquencourt, F-75205 Paris, France

ARTICLE INFO

Article history:
Received 5 December 2010
Received in revised form
27 December 2011
Accepted 7 February 2012
Available online 15 March 2012

Keywords:
Component
Dependency solving
Software evolution
Package management
Open source
Competition

ABSTRACT

Maintenance of component-based software platforms often has to face rapid evolution of software components. Component *dependencies*, *conflicts*, and *package managers* with *dependency solving* capabilities are the key ingredients of prevalent software maintenance technologies that have been proposed to keep software installations synchronized with evolving component repositories. We review state-of-the-art package managers and their ability to keep up with evolution at the current growth rate of popular component-based platforms, and conclude that their dependency solving abilities are not up to the task.

We show that the complexity of the underlying upgrade planning problem is NP-complete even for seemingly simple component models, and argue that the principal source of complexity lies in multiple available versions of components. We then discuss the need of expressive languages for user preferences, which makes the problem even more challenging.

We propose to establish dependency solving as a *separate concern* from other upgrade aspects, and present *CUDF* as a formalism to describe upgrade scenarios. By analyzing the result of an international dependency solving competition, we provide evidence that the proposed approach is viable.

1. Introduction

A program that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful.

The above law of *Continuing Change* (Lehman, 1980) applies to all evolving software systems, which are deemed to be the vast majority of existing systems (Cook et al., 2006). The advent of Component-Based Software Engineering (Brown and Wallnau, 1998; Szyperski, 1998) did not affect this fundamental truth: *mutatis mutandis* continuing change also holds for component-based systems (Lehman and Ramil, 2000). The diffusion of rapidly evolving *component-intensive software platforms*—i.e. platforms where the number of components is in the tens or even hundreds of thousands—has raised the quality requirements for automatic tools that maintain component installations on behalf of users, be them developers, architects, administrators, or final users empowered to assemble components.

Component-intensive platforms are commonplace: FOSS (Free and Open Source Software) distributions (where components are called "packages"), development platforms like Eclipse and Apache Maven (Des Rivières and Wiegand, 2004; Massol and O'Brien, 2005) (which call components "plugins"), OSGi (OSGi Alliance) ("bundles"), CMS communities ("add-ons"), Web browsers ("extensions"), and countless others. Despite apparent differences in terminology, all these platforms share concepts, properties, and problems. For instance, components have expectations on the deployment context: they may need other components to function properly—declaring this fact by means of *dependencies*—and may be incompatible with some other components—declaring this fact by means of *conflicts*. Those expectations must be respected not only at initial deployment-time, but also at each component release and for each individual component: a new version of a component cannot be deployed if its expectations are not met on the target system.

To maintain component assemblies, (semi-)automatic component manager applications are used to perform component installation, removal, and upgrades on target machines—we use the term *upgrade* to refer to any combination of those actions. Examples of component managers are as commonplace as component-intensive platforms: package managers, such as APT or Aptitude used in FOSS distributions to manage packages; P2 (Le Berre and Rapicault, 2009), used in Eclipse to deal with plugins; OSGi

☆ This work has been partially performed at IRILL http://www.irill.org.

* Corresponding author.
E-mail addresses: abate@pps.jussieu.fr (P. Abate), roberto@dicosmo.org (R. Di Cosmo), treinen@pps.jussieu.fr (R. Treinen), zack@pps.univ-paris-diderot.fr (S. Zacchiroli).
URLs: http://www.pps.jussieu.fr/~abate (P. Abate), http://www.dicosmo.org (R. Di Cosmo), http://www.pps.jussieu.fr/~treinen/ (R. Treinen), http://upsilon.cc/~zack (S. Zacchiroli).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2012.02.018

```
# aptitude upgrade
1163 packages upgraded, 633 newly installed,
195 to remove and 0 not upgraded.
The following packages have unmet dependencies:
[...]
open: 4892; closed: 4995; defer: 170; conflict: 86
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 7592; closed: 7654; defer: 193; conflict: 89
open: 7798; closed: 7879; defer: 233; conflict: 89
open: 9938; closed: 9977; defer: 315; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 14915; closed: 14952; defer: 372; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 19880; closed: 19981; defer: 445; conflict: 89
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 25017; closed: 24998; defer: 467; conflict: 90
No solution found within the allotted time. Try harder? [Y/n]
Resolving dependencies...
open: 30110; closed: 29978; defer: 498; conflict: 91
No solution found within the allotted time. Try harder? [Y/n]n
```

Fig. 1. Unexpected behaviour while using the legacy Aptitude package manager, on a FOSS system on the Debian GNU/Linux distribution. The user attempts to upgrade all components in need of upgrade on a machine equipped with the GNOME desktop environment and several LaTeX packages. The dependency solver loops and is unable to find a solution; after several attempts, the user gives up (see http://bugs.debian.org/590470; retrieved November 29th, 2010).

resolvers, which perform component deployment and configuration. These tools—called generically *package managers* in the following—incorporate numerous functionalities: trusted retrieval of components from remote repositories; planning of upgrade paths in fulfillment of deployment expectations (also known as *dependency solving*); user interaction to allow for interactive tuning of upgrade plans; and the actual deployment of upgrades by removing and adding components in the right order, aborting the operation if problems are encountered at deploy-time (Di Cosmo et al., 2008).

In contexts where the pace of component releases is rapid (e.g. FOSS Raymond, 2001; Gonzalez-Barahona et al., 2009; Abate et al., 2009) the quality demand on package managers, and in particular on dependency solving, is very high. Package managers should: (1) devise upgrade plans that are correct (i.e. no plan that violates component expectations is proposed) and complete (i.e. every time a suitable plan exists, it can be found); (2) have performances that scale up gracefully at component repositories growth; (3) empower users to express preferences on the desired component configuration when several options exist, which is often the case. Surprisingly, all mainstream component manager applications the authors are aware of fail to address one or several of those concerns. Not addressing them is far from being a purely academic exercise, as Figs. 1 and 2 show. Although anecdotal those and similar examples, which populate the experience of everyday package manager users, show that state-of-the-art component managers are short of fulfilling the aforementioned requirements. Considering the recent popularity of dependency-based abstractions in Component Based Software Engineering (CBSE, e.g. Jenson et al., 2010; Schmid, 2010; Di Cosmo and Zacchiroli, 2010), overlooking important dependency solving requirements appears to be dangerous.

This work provides substantial coverage of concepts and problems that are common in component managers equipped with automatic dependency solving abilities, for any non-trivial component model. Understanding such problems is of paramount importance because, in the context of component-intensive software platforms, software evolution is observed by users through the lens of component releases and often judged by the package manager abilities to successfully deploy new releases. Therefore, to avoid software evolution bottlenecks at the component deployment stage, we need to improve the ability of our tools to plan component upgrades. Unfortunately, as we will show, the problem is a hard one to tackle. In order to attack such a non-trivial and fairly overlooked problem, this paper proposes to treat dependency solving as a separate concern of component evolution and details the formalisms and technologies that are needed to enable such separation.

```
# aptitude install baobab
[...]
The following packages are BROKEN: gnome-utils
The following NEW packages will be installed: baobab
[...]
The following actions will resolve these dependencies:
Remove the following packages:
  gnome gnome-desktop-environment libgdict-1.0-6
Install the following packages:
  libgnome-desktop-2 [2.22.3-2 (stable)]
Downgrade the following packages:
  gnome-utils [2.26.0-1 (now) -> 2.14.0-5 (oldstable)]
[...]
0 packages upgraded, 2 newly installed, 1 downgraded,
180 to remove and 2125 not upgraded. Need to get 2442kB
of archives. After unpacking 536MB will be freed.
Do you want to continue? [Y/n/?]
```

Fig. 2. Attempt to install a disk space monitoring utility (called *baobab*) using Aptitude. In response to the request, the package manager proposes to downgrade the GNOME desktop environment all together to a very old version compared to what is currently installed. As shown in Section 6 a trivial alternative solution exists that minimizes system changes: remove a couple of dummy (or "meta") packages.

1.1. Paper contributions and structure

In Section 2 we present the upgrade planning problem, or simply *upgrade problem*, in a general setting, showing that in any non-trivial component model dependency solving is NP-complete. To tackle the problem, in Section 3 we propose to treat dependency solving as a separate concern, in order to share research and development efforts on upgrade planning. To that end, we need formalisms to: (1) capture upgrade scenarios coming from different component models in a unifying, well-defined semantics and (2) describe user preferences which are advanced enough to cover realistic use cases, but yet simple enough to be efficiently dealt with by state-of-the-art constraint solvers. Our proposals for those two formalisms are detailed in Sections 4 and 5. Section 6 validates the proposed approach by discussing an international dependency solving competition—called MISC—which has been run exploiting the proposed formalisms. Competition results show that state-of-the-art constraint solvers can easily outperform ad-hoc solvers embedded in mainstream package managers, confirming the thesis that separation of concerns and reuse are not only feasible, but also a viable strategy to improve upgrade planning and support component evolution.

2. Component evolution and the complexity of the upgrade problem

In this section we start by studying the complexity of the *upgrade problem* that package managers for component-intensive software platforms have to face. An important feature of the problem is that there is usually a multitude of possible choices. This has two consequences:

- For any given user request, there potentially exists an exponential number of solution candidates, which makes the problem NP-complete in all relevant cases (see Sections 2.1 and 2.2).
- There might be an exponential number of actual solutions to a problem instance, and we need a good way to pick the best among these solutions (see Section 2.3).