Information and Software Technology 56 (2014) 438–462

Contents lists available at ScienceDirect

Information and Software Technology

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

Simulating upgrades of complex systems: The case of Free and Open Source Software

Davide Di Ruscio, Patrizio Pelliccione*

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica (DISIM), Università dell'Aquila, Italy

A R T I C L E   I N F O

Article history:
Received 1 March 2013
Received in revised form 10 January 2014
Accepted 12 January 2014
Available online 25 January 2014

Keywords:
Evolution of FOSS systems
Simulation
Model-driven engineering
Linux distributions

A B S T R A C T

Context: The upgrade of complex systems is intrinsically difficult and requires techniques, algorithms, and methods which are both expressive and computationally feasible in order to be used in practice. In the case of FOSS (Free and Open Source Software) systems, many upgrade errors cannot be discovered by current upgrade managers and then a system upgrade can potentially lead the system to an inconsistent and incoherent state.
Objective: The objective of this paper is to propose an approach to simulate the upgrade of complex systems in order to predict errors before they affect the real system.
Method: The approach promotes the use of model-driven engineering techniques to simulate the upgrade of complex systems. The basic idea is to have a model-based description of the system to be upgraded and to make use of model transformations to perform the upgrade on a source model so to obtain a target model representing the state of the upgraded system.
Results: We provide an implementation of the simulator, which is tailored to FOSS systems. The architecture of the simulator is distribution independent so that it can be easily instantiated to specific distributions. The simulator takes into account also pre and post-installation scripts that equip each distribution package. This feature is extremely important since maintainer scripts are full-fledged programs that are run with system administration rights.
Conclusions: The paper shows the kind of errors the simulator is able to predict before upgrading the real system, and how the approach improves the state of the art of package managers while integrated in real Linux distribution installations.

1. Introduction

Modern software systems are moving towards an on-line mode of operation where long downtimes due to maintenance problems are no more acceptable. With the shift towards an on-line mode of operation the "minimal acceptable standard" for the quality of service of modern software systems has been raised to a very high level.

Free and Open Source Software (FOSS) systems are among the most challenging modern software systems. Even big companies such as Google, or large public bodies, like the French Ministry of Finance, base their information technology infrastructures on FOSS components. FOSS systems are typically based on fine-grained units of software deployment, called packages, which evolve in a non-centralized and controlled way and are frequently released [1].

The complexity of deploying a complete FOSS infrastructure has led to the development of what are now called *distributions* (e.g., Mandriva, Ubuntu, and Fedora), which are consistent and functional sets of software components released together with the software that is necessary to set up a complete operating system. This software includes also automated mechanisms for managing the packages distributions are made of. These automated mechanisms, called package managers, make use of packages metadata in order to upgrade the system, i.e., to handle package installations, removals, and upgrades. The most important information that is contained in packages metadata concerns the specification of dependencies (i.e., what a package needs in order to be correctly installed and to function correctly), and conflicts (i.e., what should not be present on the system in order to avoid malfunctioning). These packages are equipped with maintainer scripts which are executed before and after upgrades to perform configuration actions. Unfortunately, by using existent package managers it is possible to easily make the system unstable by installing, removing, or upgrading some packages that "break" the consistency and coherence of what is installed in the system itself.

* Corresponding author. Tel.: +39 3284356799.
E-mail addresses: davide.diruscio@univaq.it (D. Di Ruscio), patrizio.pelliccione@univaq.it (P. Pelliccione).

0950-5849/$ - see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2014.01.006

The Mancoosi project[1] investigated the problem of ensuring the "correct" upgrade of FOSS systems. This is a complex problem that goes beyond the installation of single packages since it has to deal also with the preservation of some additional properties and with finding an "optimal-path" to migrate the system from the current configuration to the targeted new one. In the context of the Mancoosi project, we have conceived Evoss (EVolution of free and Open Source Software) [2], which is a model-based approach to support the upgrade of FOSS systems. The approach promotes the simulation of upgrades to predict failures before affecting the real system. Both fine-grained static aspects (e.g., configuration incoherences) and dynamic aspects (e.g., the execution of configuration scripts) are taken into account, improving over the state of the art of package managers.

While a good introduction to the overall Evoss approach and its motivations can be found in [2], this paper describes in details the simulator by presenting its structure, its constituting elements, its implementation and by discussing the kind of errors the simulator is able to predict before they affect the real system. The approach promotes the use of model-driven engineering techniques to simulate the upgrade of complex systems. The basic idea is to have a model-based description of the system to be upgraded and to make use of model transformations to perform the upgrade on a source model so to obtain a target model representing the state of the upgraded system. The implementation of the approach is focused on FOSS systems. The architecture of the simulator is distribution independent so that it can be easily instantiated to specific distributions (each distribution has some specificities such as shell commands and conventions used to write maintainer scripts). The simulator takes into account also pre and post-installation scripts that equip each distribution package. This feature is extremely important since maintainer scripts are full-fledged programs that are run with system administration rights. The simulation of maintainer scripts is possible thanks to a Domain Specific Language (DSL), which has been conceived within Evoss to specify maintainer scripts behavior. The DSL captures recurring templates and a limited set of control flow operations, which have been carefully identified through a deep analysis of Linux distributions. Moreover, the DSL has also a tagging mechanism that allows us to specify the behavior of those parts of scripts that cannot be completely specified with DSL statements. This way, script authors (usually package maintainers) can specify how such parts affect the configuration model and enable their simulation. DSL elements have a well-defined transformational semantics expressed in terms of system configuration modifications: each system configuration is given as a model and the script behavior is represented by suitable model transformations that take as input a system model representing the system before performing the installation.

Real experiences show how the simulator works in practice by highlighting its efficacy, i.e., how it improves the state of the art of package managers while integrated in real Linux distribution installations. The overall approach has been implemented by Caixa Mágica[2] within the distribution CM14.[3] This distribution provides to users also a selective roll-back mechanism that enables roll-backs also of single operations. The effect of the revert operation is the system configuration in which only the selected operations have been roll-backed. This is completely different from currently available roll-back systems that allow a previous configuration of a system to be restored, but any changes between that time and the current time, even if they affect other configuration files, are lost.

Paper structure. Section 2 introduces the case of FOSS systems and discusses limitations of existent package managers. Section 3 presents the simulator, which is the main contribution of the paper. Section 4 describes how the simulator has been implemented by using model-driven technologies. The simulator can be integrated with current Linux distributions and this is presented in Section 5. Section 6 mainly discusses the actual upgrade problems EVOSS is able to deal with, and presents a large experimentation, which has the aim of validating the simulator and of showing its applicability in practice. Section 7 compares the approach with related works. Section 8 briefly presents the history of simulation in computer science and discusses the benefits of using model-driven technologies for realizing simulators. The paper concludes in Section 9 by providing final remarks and future research directions.

2. FOSS distributions

Widely used FOSS distributions, like Debian, Ubuntu, Fedora, and Suse are based on the central notion of software *package*. Packages are assembled to build a specific software system. The recommended way of evolving such systems is to use *package manager* tools to perform system modifications by adding, removing, or replacing packages. These operations are generically called *upgrades*. Existent package managers integrate some preliminary checks to prevent upgrade failures. Unfortunately many unpredicted upgrade failures might still happen, as will be discussed in the following.

2.1. Packages and upgrades

In FOSS distributions, a *package* is a software unit containing the software component and a description of it, also known as *metadata*. More precisely, the core of each package is a *file bundle* encoding the shipped component: executable binaries, data, documentation, etc. A distinguished subset of those files consists of *configuration files*, which affect the runtime behavior of the component. During upgrade deployment, most files play a "static" role, in the sense that they are simply copied over.

Package metadata describe aspects needed for calculating an *upgrade plan* according to upgrade requests. An upgrade plan is produced by a planner (which may consider also optimization criteria) and consists of a sequence of packages with associated the operation to be performed (i.e., installation and removal of a package). Common metadata contain the package identifier, version, maintainer, and description. Most notably, metadata are also used to declare *inter-package relationships* such as: dependencies (the need of other packages to work properly), conflicts (the incompatibilities with other packages), and feature provisions (an indirection layer over dependencies) [3]. This information is taken into account by the package manager when performing the system upgrade.

Packages also contain a set of executable *maintainer scripts*, used by package maintainers to hook custom actions into the upgrade process. Several aspects of maintainer scripts are noteworthy:

1. they play a dynamic role as they are executed *during* upgrades;
2. they are full-fledged programs, usually written in POSIX shell languages;
3. they are executed with system administrator rights and then they may perform arbitrary changes to the whole system;
4. they cannot be replaced by just shipping extra files: they might need to access data which is available in the target installation machine, but not in the package itself;
5. they are expected to complete without errors: their failures, usually signaled by non-0 exit codes, automatically trigger upgrade failures.

[1] FP7 EU project Mancoosi: http://www.mancoosi.org.
[2] http://www.caixamagica.pt/.
[3] http://linux.caixamagica.pt/pag/documentacao/CM14/ManualCxM14.pdf.