Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 17–32
www.elsevier.com/locate/entcs

Dependency Management in Software Component Deployment

Meriem Belguidoum[1] and Fabien Dagnat[2]

Department of Computer Sciences
ENST Bretagne, Technopole Brest-Iroise
Brest, France

Abstract

Component-based distributed systems are hard to deploy for two main reasons: the complexity of their structure and the complexity of the deployment tasks. Current tools do not manage these complexities properly because the descriptions that they allow lack of expressiveness. The absence of proper descriptions of system and component requirements makes it impossible to ensure safe installation and deinstallation. The goal of this paper is to present a formalization of deployment dependencies. These dependencies expressed in a logical language are associated with a deployment engine that allows installation and deinstallation of components in a system to be proved.

Keywords: Component deployment, Deployment dependencies, Safe deployment.

1 Introduction

The component approach to building systems is gaining audience because of the interesting properties of components. We can imagine that software will soon be very large collections of components and that the reuse and sharing of components will be common practice. However, components are often developed by different groups and their dependencies are not clearly specified. Hence installing (or deinstalling) a component is often a gamble since all the dependencies are difficult to find. Using current approaches, installation may not achieve *success* [15] (an installed component does not work) and installation or deinstallation may not be *safe* and disrupt the system. To face the evolution towards component based systems, our aim is to build a tool with formal foundations ensuring the success and safety of deployment.

In this paper, we present the formalization of a static deployment system that ensures the success and safety of installation and deinstallation. What is meant by

[1] Email: meriem.belguidoum@enst-bretagne.fr
[2] Email: fabien.dagnat@enst-bretagne.fr

doi:10.1016/j.entcs.2006.09.029

static is that we do not address the dynamic reconfiguration of the interconnection in the system yet. The work presented here does not take into account the concrete realization of the deployment operations. We only present a reasoning framework to authorize or forbid deployment. Furthermore, our work is based on the fact that components come with an exact [3] description of their requirements and effect on the system.

The central concept is the notion of *dependency* that abstracts the link between component and hardware requirements. Dependencies are used during installation to ensure the requirements are fulfilled and during deinstallation to guarantee that they still be fulfilled. This paper defines dependencies, how they are specified and describes how to properly manage them during installation and deinstallation.

This paper is organized as follows. First, section 2 introduces the concept of component deployment and illustrates deployment dependencies using the example of a mail server. Next, section 3 presents our description of dependency deployment and section 4 the description of environmental constraints. Then, in section 5 and section 6 we present a formalization of the installation and deinstallation of components and the management of their effect on the target system respectively. Finally, we discuss related work in section 7 and conclude this article by presenting some future work in section 8.

2 Component deployment

In this paper, we use a basic abstract notion of component. A component provides services that require services. Components and services are identified by their names. A required service is specified giving its name (and possibly the name of its provider). This work can be applied to any component model supporting named services and components and the notions of required and provided services.

Notice that using names to specify requirements would require to have some component and service dictionnary. In an open setting this requirement can be a limitation but is already used in practice by the packaging systems of Linux. To overcome this limitation, work is needed to enable the use of other form of identity for services and components (such as, for example, interface types for services).

A component software is a set of interconnected components. This interconnection is the software architecture and is often designed using an Architecture Description Language (ADL) such as Fractal ADL [3] or xADL [10]. Fig. 1 illustrates such an architecture for a mail server on a Linux system. It is composed of four components: *Postfix*, an SMTP server playing the role of a Mail Transport Agent (MTA), *Fetchmail* that allows to recover mail by an electronic mail transport protocol (*e.g.*, Pop) from a distant host (the messages are redirected to the local transport), *Procmail*, a Mail Deliver Agent (MDA) that manages received mails and allows, for example, the filtering of a mail. Finally, *Sylpheed*, a mail manager for reading and composing mail called a Mail User Agent (MUA).

[3] This means that the component does not hide requirements or effects.

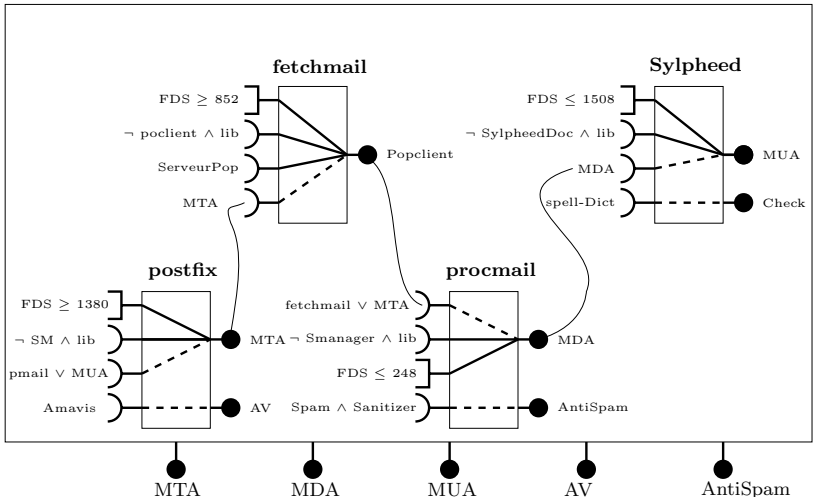

Fig. 1. Assembly of the components relating to the mail server

The installation/deinstallation of a component in/from a system corresponds to its addition/removal to/from the system. The success of these actions requires that:

(i) The system provides the resources and the services required by the component (being installed).

(ii) The component (being installed) or one of its services do not conflict with already installed services.

(iii) The services provided by the component (being deinstalled) are not used by other components in the system.

To answer these questions, we need (1) the resources of the target system (2) its architecture and (3) the component description.

The *resources* are here abstracted by a set of environment variables. We suppose that a standard choice of names and valuations for the resource description (for example the Management Information Format [5]) is made. The values are obtained by sensors and therefore will not be modified by our rules.

The *architecture* of the system memorizes the interconnection between all components in the system. Such a complete explicit architecture (if it exists) would be unmanageable because of its size. Furthermore, constructing it can be very difficult as dependencies are often partly hidden. Therefore, rather than supervising all the installation and deinstallation scripts, we advocate the use of an approximation of the *inter-dependencies* between components. This approximation is discussed in greater detail in section 4.

The *description of a component* must be sufficiently precise to express the links of a provided service to its requirements. This gray-box description specifies *intra-dependencies* which are parametrized contracts [17], that is, outputs (provided services) are linked to the entries (required services) they depend on. The form of these links is defined in the next section.

3 Dependency specification

In this section, we present the precise definition of the relation between a required and a provided service (either of the same component or of two components). Such a relation is called a dependency. The mail server example already introduced illustrates dependencies in Fig. 1 [4]. There is three main forms of dependencies, a dependency is either *mandatory*, *optional* or *negative*:

- a mandatory dependency (represented by a solid line) is a firm requirement. If it is not fulfilled installation is not possible. For example, the mail server needs a terminal with a specific CPU or specific libraries, etc.
- an optional dependency (represented by a dotted line) specifies that the component may provide optional services. Such services may not be provided (if their requirements are not fulfilled) without preventing the installation. For example, `postfix` may provide a service for scanning messages against viruses if the service `Amavis` is available. Otherwise `postfix` can be installed and provides the `MTA` service, but the service `AV` is not provided.
- a negative dependency (expressed by a negation) specifies a conflict forbidding installation. The conflict may hold with a service or a component. For example, `postfix` cannot be installed if another `MTA` is already installed (such as `sendmail` for example).

The (intra-)dependency description language [5] uses the concepts of *dependency* and *predicate* defined by the following grammar where s represents the name of a service and c the name of a component:

$$D ::= P \Rightarrow s \mid D \bullet D \mid D \sharp D \mid ?\, D \qquad P ::= true \mid P \wedge P \mid Q$$

$$Q ::= Q \vee Q \mid R \qquad R ::= [v\ O\ val] \mid \neg\, s \mid \neg\, c \mid c.s \mid s \qquad O ::= > \mid \geq \mid < \mid \leq \mid = \mid \neq$$

The precise semantics of these operators will be defined by the installability and installation rules (resp. Fig. 4 and Fig. 6). Intuitively, a dependency may be the conjunction $\bullet$ or the disjunction $\sharp$ of two dependencies, an optional dependency $?$ or a simple dependency $P \Rightarrow s$ specifying the requirements P of a service s. The requirements are expressed in a first order predicate language in conjunctive normal form to simplify the installation rules. The five raw conditions (R) express a comparison on the value of an environment variable $[v\ O\ val]$, a conflict with a service $\neg\, s$ or a component $\neg\, c$ or the requirement of a service provided by a precise component $c.s$ or any component s. Examples of such predicates appear in Fig. 1 on the required interfaces (left hand side).

It is important to notice that a component may forbid a service it provides. This feature can be used by a component providing s to forbid the (future) installation of any other component providing s ($\neg\, s \Rightarrow s$).

[4] The dependencies are simplified compared to the real case.

[5] A more human friendly language exists but is not in the scope of this paper.

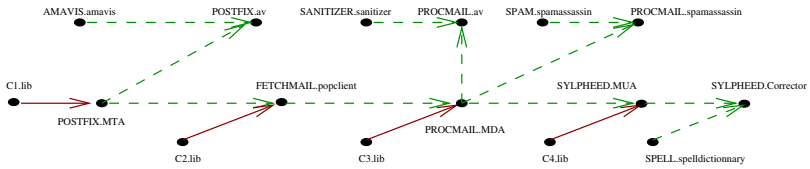

Fig. 2. Dependency graph of the mail server of Fig. 1

4 Context description

The resources and the architecture of the target system are modeled by the notion of context. Ideally, it could be the union of the dependencies of all components (part of the system). But, the calculation (and the manipulation) of this union is not realistic. Thus, a safe approximation (of this union) is needed. To ensure safe installation, we have to know the available services (with their providers) and installed components to check services' requirements and conflicts. We also need the values of the environment variables. For safe deinstallation, we need to keep all potential dependencies between services.

In this paper, the *Context* is composed of (1) an environment $\mathcal{E}$ storing the values of variables, (2) a set $\mathcal{C}$ of four-tuples $(c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ storing for each installed component c its provided services $\mathcal{P}_s$, forbidden services $\mathcal{F}_s$ and forbidden components $\mathcal{F}_c$ and (3) a dependency graph $\mathcal{G}$ storing the dependencies. A node of $\mathcal{G}$ is an available service and its provider $(c.s)$ and an edge is a pair of nodes $n_1 \mapsto n_2$ meaning that n_2 requires n_1. Each edge is labeled (above the arrow) by the kind of dependency, either mandatory M or optional O. Fig. 2 presents the dependency graph of the mail server of Fig. 1. A dependency graph is defined by the set of its labeled edges. It is built during installation and used during deinstallation. This means that an edge $n_1 \mapsto n_2$ in $\mathcal{G}$ denotes that n_2 is available and requires n_1. It implies that n_1 was available before n_2, that is the graph does not contain cycles. In practice this can be a limitation because two components may be mutually dependent. We think that such circularity should be solved [6] by building composite components that hides this circularity to the system. This composition operation is not yet available and left for future work.

To simplify the presentation of our rules, let us define available / forbidden services and components [7]:

$$\begin{cases} AS(Ctx) = \bigcup\{\mathcal{P}_s \mid (-, \mathcal{P}_s, -, -) \in Ctx.\mathcal{C}\} \\ AC(Ctx) = \{c \mid (c, -, -, -) \in Ctx.\mathcal{C}\} \\ FS(Ctx) = \bigcup\{\mathcal{F}_s \mid (-, -, \mathcal{F}_s, -) \in Ctx.\mathcal{C}\} \\ FC(Ctx) = \bigcup\{\mathcal{F}_c \mid (-, -, -, \mathcal{F}_c) \in Ctx.\mathcal{C}\} \end{cases}$$

[6] Indeed, it introduces a high coupling forbidding the separate installation or deinstallation.

[7] A dotted notation is adopted in this paper to access a specific member of a tuple directly and $-$ is used as a joker matching anything.

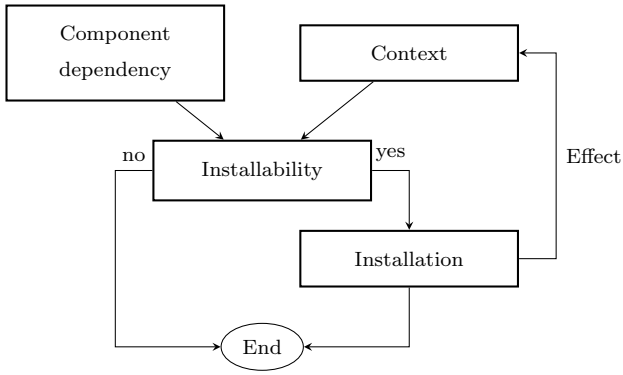

Fig. 3. Installation phases

5 Safe installation

In our approach, abstract installation is carried out in two stages (see Fig. 3). First we check whether installation is possible (*installability*) by evaluating the component dependency in the current context. Then if installation is possible, we calculate its effect on the context. This effect is used to update the abstract context once the concrete installation has been carried out.

5.1 Installability

Before authorizing the installation of a component, we have to ensure (1) it is not forbidden, (2) the services it requires are available in the context and (3) it does not provide forbidden services. More formally:

Definition 5.1 (Installability) A component c with a dependency D is *installable* within a context Ctx ($Ctx \vdash c : D$) iff the component is not forbidden and D is verified by the checking rules of Fig. 4:

$$\text{CComp:}\ \frac{Ctx \vdash_C D \qquad c \notin FC(Ctx)}{Ctx \vdash c : D}$$

The checking rules of Fig. 4 ensure that *mandatory* dependencies of the component are verified. For a simple dependency $P \Rightarrow s$, this means that P evaluates to true and s is not forbidden (CTriv). The evaluation of a predicate P in the context Ctx follows classical logic and is presented in the first part of the figure (rules $Ctx \vdash_P P$). During this stage, optional dependencies are ignored (COpt) because such dependencies may be unavailable without preventing component installation. The conjunction of dependencies is resolved when the two dependencies are valid (CAnd) and their disjunction when one of the two dependencies is valid (COrL and COrR).

Predicates:

$$\textsc{PTrue}: Ctx \vdash_P true \qquad \textsc{PAnd}: \frac{Ctx \vdash_P Q_1 \qquad Ctx \vdash_P Q_2}{Ctx \vdash_P Q_1 \wedge Q_2}$$

$$\textsc{POrL}: \frac{Ctx \vdash_P R_1}{Ctx \vdash_P R_1 \vee R_2} \qquad \textsc{POrR}: \frac{Ctx \vdash_P R_2}{Ctx \vdash_P R_1 \vee R_2} \qquad \textsc{PVar}: \frac{Ctx.\mathcal{E}(v)\ O\ V}{Ctx \vdash_P [v\ O\ V]}$$

$$\textsc{PNotS}: \frac{s \notin AS(Ctx)}{Ctx \vdash_P \neg\, s} \qquad \textsc{PNotC}: \frac{c \notin AC(Ctx)}{Ctx \vdash_P \neg\, c} \qquad \textsc{PServ}: \frac{s \in AS(Ctx)}{Ctx \vdash_P s}$$

$$\textsc{PComp}: \frac{(c, \mathcal{P}_s, -, -) \in Ctx.\mathcal{C} \qquad s \in \mathcal{P}_s}{Ctx \vdash_P c.s}$$

Dependencies:

$$\textsc{CTriv}: \frac{Ctx \vdash_P P \qquad s \notin FS(Ctx)}{Ctx \vdash_C P \Rightarrow s} \qquad \textsc{CAnd}: \frac{Ctx \vdash_C D_1 \qquad Ctx \vdash_C D_2}{Ctx \vdash_C D_1 \bullet D_2}$$

$$\textsc{COpt}: Ctx \vdash_C ?\, D \qquad \textsc{COrL}: \frac{Ctx \vdash_C D_1}{Ctx \vdash_C D_1 \,\#\, D_2} \qquad \textsc{COrR}: \frac{Ctx \vdash_C D_2}{Ctx \vdash_C D_1 \,\#\, D_2}$$

Fig. 4. Installability rules

5.2 Installation

Once the component is proved to be installable, we need to calculate the effect of its installation on the system. This effect consists of new available services, new forbidden services, new forbidden components and a new dependencies (represented by a dependency graph). Before giving the installation rules, we will show how this effect is calculated by defining two operations: $CalcF$ that determines forbidden services and components and the dependency graph calculation.

First, the services and components forbidden by a component are calculated by collecting negatives of the predicates of its dependency. This is done by the function $CalcF$ defined below.

The only case that deserve discussion is the disjunction. Indeed, several subterms of a disjunction may forbid services (or components). For example, in the dependency expression $\neg a \vee \neg b \Rightarrow S$, a or b could be forbidden. To keep track of this possibilities a complex system can be built that will record which negatives are (really) needed. Here if a is available (resp. b) we could keep $\neg\, b$ (resp. $\neg\, a$) and while none of them is available we keep the disjunction. We have chosen to present here a simpler system because we think that the benefit in terms of precision is not worth its cost. That is all services and components with negative predicates in a disjunction are forbidden (the set of forbidden services in the case of disjunction are the same as in that of conjunction).

$$\textsc{GTrue}: Ctx, c, s \vdash_G true \Rightarrow \varnothing$$

$$\textsc{GAnd}: \frac{Ctx, c, s \vdash_G Q_1 \Rightarrow \mathcal{G}_1 \qquad Ctx, c, s \vdash_G Q_2 \Rightarrow \mathcal{G}_2}{Ctx, c, s \vdash_G Q_1 \wedge Q_2 \Rightarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\textsc{GOr}: \frac{Ctx, c, s \vdash_G R_1 \Rightarrow \mathcal{G}_1 \qquad Ctx, c, s \vdash_G R_2 \Rightarrow \mathcal{G}_2}{Ctx, c, s \vdash_G R_1 \vee R_2 \Rightarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\textsc{GVar}: Ctx, c, s \vdash_G [v\ O\ V] \Rightarrow \varnothing \qquad \textsc{GNotS}: Ctx, c, s \vdash_G \neg\, s' \Rightarrow \varnothing$$

$$\textsc{GNotC}: Ctx, c, s \vdash_G \neg\, c' \Rightarrow \varnothing \qquad \textsc{GServC}: \frac{(c', \mathcal{P}_s, -, -) \in Ctx.\mathcal{C} \qquad s' \in \mathcal{P}_s}{Ctx, c, s \vdash_G c'.s' \Rightarrow \{c'.s' \overset{\mathsf{M}}{\longmapsto} c.s\}}$$

$$\textsc{GServ}: \frac{s' \in AS(Ctx)}{Ctx, c, s \vdash_G s' \Rightarrow \{c'.s' \overset{\mathsf{M}}{\longmapsto} c.s \mid (c', \mathcal{P}_s, -, -) \in Ctx.\mathcal{C} \wedge s' \in \mathcal{P}_s\}}$$

Fig. 5. Graph calculation rules

Definition 5.2 (CalcF) The function $CalcF$ calculates the set of forbidden services and the set of forbidden components from a predicate:

$$\begin{cases} CalcF(true) = \varnothing, \varnothing \\ CalcF(Q_1 \wedge Q_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \text{ where } CalcF(Q_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ CalcF(R_1 \vee R_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \text{ where } CalcF(R_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ CalcF(s) = CalcF(c.s) = CalcF([v\ O\ V]) = \varnothing, \varnothing \\ CalcF(\neg\, s) = \{s\}, \varnothing \\ CalcF(\neg\, c) = \varnothing, \{c\} \end{cases}$$

The dependency graph is built during the installation phase using the context and the service provided by the component being installed. For this, the dependency graph collects all dependencies added by the component.

Definition 5.3 (Graph calculation) The dependency graph $\mathcal{G}$ introduced by a component c when providing a service s in the context Ctx is calculated from its predicate P ($Ctx, c, s \vdash_G P \Rightarrow \mathcal{G}$) by the rules of Fig. 5.

The only rules causing new dependencies are those specifying service requirements. The rule GServ adds a dependency between each potential provider of a service and the service requiring it. The rule GServC ensures that c' provides s' and produces the corresponding dependency.

Lastly, the installation is defined by:

$$\textsc{ITriv}: \frac{Ctx, c, s \vdash_G P \Rightarrow \mathcal{G} \qquad s \notin FS(Ctx) \qquad CalcF(P) = \mathcal{F}_s, \mathcal{F}_c}{Ctx, c \vdash_I (P \Rightarrow s) \Rightarrow \{s\}, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}$$

$$\textsc{INot}_1: \frac{Ctx \vdash_P P}{Ctx, c \vdash_I (P \Rightarrow s) \Rightarrow \bot} \qquad \textsc{INot}_2: \frac{s \in FS(Ctx)}{Ctx, c \vdash_I (P \Rightarrow s) \Rightarrow \bot}$$

$$\textsc{IOpt}_1: \frac{Ctx, c \vdash_I D \Rightarrow \bot}{Ctx, c \vdash_I ?\, D \Rightarrow \varnothing, \varnothing, \varnothing, \varnothing}$$

$$\textsc{IOpt}_2: \frac{Ctx, c \vdash_I D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{Ctx, c \vdash_I ?\, D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \{s \overset{\mathsf{O}}{\longmapsto} s' \mid s \overset{-}{\longmapsto} s' \in \mathcal{G}\}}$$

$$\textsc{IAnd}_1: \frac{Ctx, c \vdash_I D_1 \Rightarrow \bot}{Ctx, c \vdash_I D_1 \bullet D_2 \Rightarrow \bot} \qquad \textsc{IAnd}_2: \frac{Ctx, c \vdash_I D_2 \Rightarrow \bot}{Ctx, c \vdash_I D_1 \bullet D_2 \Rightarrow \bot}$$

$$\textsc{IAnd}_3: \frac{Ctx, c \vdash_I D_1 \Rightarrow \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1, \mathcal{G}_1 \qquad Ctx, c \vdash_I D_2 \Rightarrow \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2, \mathcal{G}_2}{Ctx, c \vdash_I D_1 \bullet D_2 \Rightarrow \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\textsc{IOrL}: \frac{Ctx, c \vdash_I D_1 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{Ctx, c \vdash_I D_1 \# D_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_c, \mathcal{F}_s, \mathcal{G}}$$

$$\textsc{IOrR}: \frac{Ctx, c \vdash_I D_1 \Rightarrow \bot \qquad Ctx, c \vdash_I D_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{Ctx, c \vdash_I D_1 \# D_2 \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}$$

Fig. 6. Installation rules

Definition 5.4 (Installation) The installation of a component c with a dependency D in a context Ctx has four effects: provided services $\mathcal{P}_s$, forbidden services $\mathcal{F}_s$, forbidden components $\mathcal{F}_c$ and dependencies (graph $\mathcal{G}$). These effects are obtained by the rules of Fig. 6.

$$\textsc{IComp}: \frac{Ctx, c \vdash_I D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}{Ctx \vdash_I c : D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}$$

The effect of $P \Rightarrow s$ is undefined if either P is false ($\textsc{INot}_1$) or s is forbidden ($\textsc{INot}_2$). Otherwise, s is available, forbidden services and components are calculated by $CalcF$ and the graph by the rules of Fig. 5 ($\textsc{ITriv}$). An optional dependency $?\, D$ has almost the same effect as D if it is defined ($\textsc{IOpt}_1$), and the dependencies of D are converted to optional. Otherwise it has no effect ($\textsc{IOpt}_2$). In a conjunction $D_1 \bullet D_2$, D_1 and D_2 must be valid and then the effect is the union of their effects ($\textsc{IAnd}_3$). Otherwise it is undefined ($\textsc{IAnd}_1$ and $\textsc{IAnd}_2$). Lastly, the effect of a disjunction $D_1 \# D_2$ is that of D_1 ($\textsc{IOrL}$) if D_1 is verified, or that of D_2 ($\textsc{IOrR}$) in the opposite

case. Notice that the disjunction has the semantics of an if, the second dependency is used only if the first is not verified.

In this paper, we consider that our formal reasoning engine does not take care of updating environment variables. A concrete deployment engine updates the physical context and sensors bring it to the formal engine.

5.3 An example of installation

Let's illustrate the installation of the component `postfix` whose dependency is $([FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}) \bullet ?(S_{Amavis} \Rightarrow S_{AV})$ in a system having the description $\{(FDS = 500000)\}, \{(C_1, S_{lib}, \varnothing, \varnothing), (C_2, S_{Amavis}, \varnothing, \varnothing)\}, \varnothing$.

The installability of `postfix` is deduced by the proof presented in the first part of Fig. 7. This proof ensures that libraries are present, `sendmail` is not present, the free disk size (FDS) is bigger than the required one and the provided service S_{MTA} is not forbidden. Note that as the requirement for S_{Amavis} is optional, it is not explored.

As `postfix` is installable, the installation stage follows and calculates the effect of installing `postfix` (C_{PX}) by the proof in the second part of Fig. 7 (the requirement predicate of S_{MTA} is denoted P). During this phase, the optional dependency is checked to determine whether it provides services (here it contributes the S_{AV} service). After the installation of `postfix`, the MTA service (S_{MTA}) and the anti-virus (S_{AV}) are provided and the component `sendmail` (C_{SM}) becomes forbidden. The dependency graph $\mathcal{G}$ corresponds to the union of the dependency graphs deduced from the two sub-dependencies, that is $\mathcal{G} = \{C_2.S_{Amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}, C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}$. Therefore, after the installation of `postfix` the context becomes:

$$\begin{cases} \{(FDS = 500000)\}, \\ \{(C_1, S_{lib}, \varnothing, \varnothing), (C_2, S_{Amavis}, \varnothing, \varnothing), (C_{PX}, \{S_{MTA}, S_{AV}\}, \varnothing, \{C_{SM}\})\}, \\ \{C_2.S_{Amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}, C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\} \end{cases}$$

6 Safe deinstallation

Deinstallation of a component c is also carried out in two stages. First, we check its feasibility by ensuring (using the dependency graph) that no service provided by c is required by another component. Then, we calculate the effect of deinstallation, that is, the removal of c from the context and of edges relating to the services that c provides in the dependency graph.

To manage deinstallation, we use the dependency graph built during installation. A component can be removed, if none of its provided services are used by other components. Therefore, for each provided service, we have to check that no (mandatory) service of another component requires it. Thus, a service can be removed if either it is not used (i.e., it is a leaf of the dependency graph) or it is only

Installability:

$$\dfrac{\dfrac{500000 \geq 1380}{Ctx \vdash_P FDS \geq 1380} \quad \dfrac{C_{SM} \notin \{C_1, C_2\}}{Ctx \vdash_P \neg C_{SM}} \quad \dfrac{S_{lib} \in \{S_{lib}, S_{Amavis}\}}{Ctx \vdash_P S_{lib}}}{\dfrac{\dfrac{Ctx \vdash_P [FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \quad S_{MTA} \notin \varnothing}{Ctx \vdash_C [FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}} \quad Ctx \vdash_C ?(S_{Amavis} \Rightarrow S_{AV})}{Ctx \vdash_C ([FDS \geq 1380] \wedge \neg C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}) \bullet ?(S_{Amavis} \Rightarrow S_{AV})}}$$

Installation:

$$\dfrac{\dfrac{\dfrac{S_{lib} \in \{S_{lib}, S_{Amavis}\}}{Ctx, C_{PX}, S_{MTA} \vdash_G S_{lib} \Rightarrow \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}} \quad Ctx, C_{PX}, S_{MTA} \vdash_G \neg C_{SM} \Rightarrow \varnothing \quad Ctx, C_{PX}, S_{MTA} \vdash_G [FDS \geq 1380] \Rightarrow \varnothing}{Ctx, C_{PX}, S_{MTA} \vdash_G P \Rightarrow \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}} \quad S_{MTA} \notin \varnothing \quad CalcF(P) = \varnothing, \{C_{SM}\}}{Ctx, C_{PX} \vdash_I (P \Rightarrow S_{MTA}) \Rightarrow \{S_{MTA}\}, \varnothing, \{C_{SM}\}, \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{S_{Amavis} \in \{S_{lib}, S_{Amavis}\}}{Ctx, C_{PX}, S_{AV} \vdash_G S_{Amavis} \Rightarrow \{C_2.S_{Amavis} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{AV}\}} \quad S_{AV} \notin \varnothing \quad CalcF(S_{Amavis}) = \varnothing, \varnothing}{Ctx, C_{PX} \vdash_I (S_{Amavis} \Rightarrow S_{AV}) \Rightarrow \{S_{AV}\}, \varnothing, \varnothing, \{C_2.S_{Amavis} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{AV}\}}}{Ctx, C_{PX} \vdash_I ?(S_{Amavis} \Rightarrow S_{AV}) \Rightarrow \{S_{AV}\}, \varnothing, \varnothing, \{C_2.S_{Amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}\}}}{Ctx, C_{PX} \vdash_I (P \Rightarrow S_{MTA}) \bullet ?(S_{Amavis} \Rightarrow S_{AV}) \Rightarrow \{S_{MTA}, S_{AV}\}, \varnothing, \{C_{SM}\}, \mathcal{G}}$$

Fig. 7. Installability and Installation proofs of `postfix`

required (directly or indirectly) by optional services (in the graph, all paths coming from it must be composed of green arcs).

Definition 6.1 (Mandatory dependencies (MD)) The set of mandatory dependencies (MD) of a service s provided by a component c in a dependency graph is defined as follows:

$$MD(\mathcal{G}, c.s) = \bigcup \{\{c'.s'\} \cup MD(\mathcal{G}, c'.s') \mid c.s \overset{\mathsf{M}}{\longmapsto} c'.s' \in \mathcal{G}\}$$

Definition 6.2 (Deinstallability) A component c can be removed from a context Ctx iff all its provided services has no mandatory dependencies:

$$\textsc{Check-DI:} \quad \dfrac{(c, \mathcal{P}_s, -, -) \in Ctx.\mathcal{C} \qquad \bigcup \{MD(\mathcal{G}, c.s) \mid s \in \mathcal{P}_s\} = \varnothing}{Ctx \vdash_D c}$$

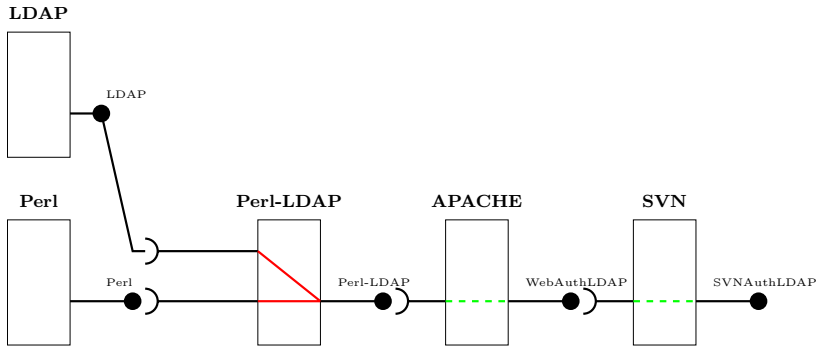

Fig. 8. An example of a dependency graph

The effect of the deinstallation of a component c on a context Ctx involves the set of nodes that must be removed from the dependency graph. This set of nodes contains all provided services of c and all (optional) services depending on them. Once the concrete deinstallation is carried out, Ctx will be updated by removing c (and its provided services, forbidden services and forbidden components) from $\mathcal{C}$ and removing [8] from $\mathcal{G}$ all nodes of the effect.

Definition 6.3 (Optional dependencies (OD)) The set of optional dependencies (OD) of a service s provided by a component c in a dependency graph is defined as follows:

$$OD(\mathcal{G}, c.s) = \bigcup\{\{c'.s'\} \cup OD(\mathcal{G}, c'.s') \mid c.s \overset{o}{\longmapsto} c'.s' \in \mathcal{G}\}$$

Definition 6.4 (Deinstallation) The deinstallation of a component c has the following effect:

$$\textsc{Effect:}\ \frac{(c, \mathcal{P}_s, -, -) \in Ctx.\mathcal{C}}{Ctx \vdash_E c \Rightarrow \bigcup\{\{c.s\} \cup OD(Ctx.\mathcal{G}, c.s) \mid s \in \mathcal{P}_s\}}$$

Let us illustrate the deinstallation of components having optional dependencies via an example. Suppose we want to have a subversion server `SVN` using LDAP authentication through `Apache` and `Perl`. The component `LDAP-Perl` allows the authentication of the user, based on the attributes of the component `LDAP`. These components must be installed in a precise order (see Fig. 8). First, `LDAP` and `Perl` are installed to enable the installation of the component `LDAP-Perl`. This component provides the service $S_{LDAP-Perl}$ optionally used by `APACHE` to provide an authentication service $S_{WebAuthLDAP}$. Finally, `SVN` can use this service to provide its own authentication service $S_{SVNAuthLDAP}$.

Let us examine the deinstallation of the component `LDAP-Perl` and thus the removal of the service $S_{LDAP-Perl}$. According to the deinstallability definition 6.2, we need to determine MD. On the left hand side of Fig. 9, we can see that in

[8] $\mathcal{G} \setminus N = \{n_1 \overset{-}{\longmapsto} n_2 \mid n_1 \overset{-}{\longmapsto} n_2 \in \mathcal{G} \wedge n_1 \notin N \wedge n_2 \notin N\}$

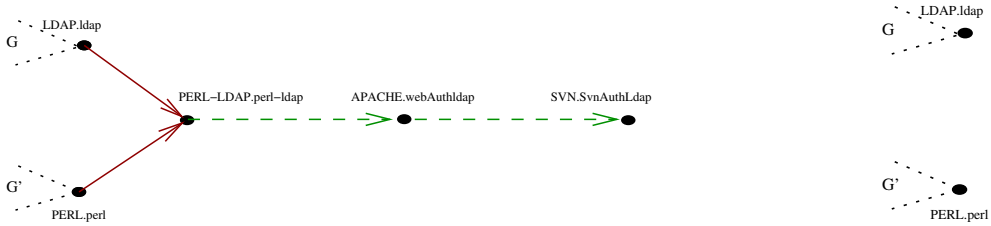

Fig. 9. Before and after removing LDAP-Perl

the dependency graph, all paths depending on this service have optional arcs. Indeed, this service is only used (optionally) by `APACHE` and then `SVN`. So, *MD* is empty and `Perl-LDAP` can be deinstalled. According to definition 6.4, to remove $S_{LDAP-Perl}$, we must remove all nodes depending on it. The calculus of *OD* gives $\{S_{WebAuthLDAP}, S_{SVNAuthLDAP}\}$. Thus, these services are removed while the components `APACHE` and `SVN` remain installed. The resulting dependency graph is shown on the right hand side of Fig 9).

7 Related work

A lot of research focuses on the description and the management of component-based systems. Deployment tools such as COACH [9] and deployment specifications such as of the OMG [13] do not support the description of deployment dependency. The constraint one may express in those framework is limited to constraints on the target environment.

In architecture description languages (ADL) [12,6], descriptions focus on the structural view and concentrate on a high level logical view of components without taking into account the physical view (real effect on physical environment). Behavioral ADL exists such as π-ADL [14] but do not address the problem of deployment. To our knowledge, [8] is the only work extending an ADL to specify deployment constraints. Their approach is to describe constraints on the location of components. These constraints enable to describe requirements on hardware, simple software dependencies and co-location. However, ignoring the problem of deinstallation they do not have to handle software dependencies. Our work aims to encompass both the logical and physical views in descriptions offering of an expressive language for the deployment constraints specifications. For this, we follow [17] using parametrized conditions to specify dependencies (i.e., the provided services differ according to available services). In Reussner paper, this approach is limited to the specification of quality attributes.

In [11] an architecture for the representation and the management of dependencies in component systems is proposed. This representation is used for component implementation, which are configured and adapted automatically to dynamic changes in the environment. In this work, dependency descriptions are assumed to be already present and consistent, while in our approach, we aim to prove the consistency of the specifications.

Lastly, little work examine safety of deployment. The EDOS project aims to manage dependencies among large collections of software packages. They build a formal system [16] to check installability. In their context, installability is a lot harder than our, because if the system does not allow a component to be installed they try to determine which minimal set of packages is necessary to enable the installation. They prove that this problem is NP-complete but show that this is not a problem in practice. Another work presented in [18] deals with the problem of software configuration management. It formalizes the package system of Debian by defining a rule-based formal language for representation of configuration knowledge. Each rule (expressing a requirement) is translated to a logic program using the stable model semantics [7]. This work focuses on this particular form of semantics rather than the management of complex dependencies.

The two last works are related to the management of software packages of the Debian linux distribution. The main difference between packages and components is the fact that a package only provide one service [9]. Furthermore a component may provide a variable number of services depending on the context. A much richer dependency language is required to take into account these two differences. This paper introduces such a language with rules to ensure the safety of installation and deinstallation.

8 Conclusion and future work

In this paper, a formalization of installation and deinstallation of components has been presented. It aims at providing a safe deployment framework that guarantee the success of installation and deinstallation. The key concept to offer this safety is the notion of dependency. A dependency abstract a components connection, it is mandatory or optional, positive (required) or negative (forbidden). The description and the management of these dependencies encompass our previous work [2,1] by extending the syntax of requirements (allowing the provider specification) and introducing the notion of dependency graph. All potential dependencies are approximated by this dependency graph (built during installation) in order to ensure safe deinstallation. A simple prototype associated to a prover has been developed in OCaml. This proof of concept prototype is currently used to test our approach on the deployment of Fractal components [4].

We are working on two main directions. First, our objective is to ensure the *guarantee of the deployment*. For this, a formalization of the properties a deployment system should respect (success, safety, ...) is needed. The goal is then to prove that our system ensure these properties. The second direction is to extends our system to overcome its current limitations. The two main limitations are:

- the deployment operations offered, a replace and an assembly operation are needed. The replace operation is needed to allow the upgrade of a component. Indeed as our system does not allow to deinstall a component providing ser-

[9] The notion of virtual package has not the same expressive power as real services.

vices used by other components, upgrading a component is not equivalent to a deinstallation and then an installation. The assembly operation is needed to calculate the dependency of a composite component using the dependencies of its sub-components.

- the component and the service identities, in our current approach names hold a central position that they should not have. The identity of a service must be extended to include interface type and version information. This means to change from name equality to a form of subtyping when determining dependencies between services.

References

[1] Belguidoum, M. and F. Dagnat, *Analysis of deployment dependencies in software components*, in: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (2006).

[2] Belguidoum, M., F. Dagnat and A. Beugnard, *Analyse des dépendances pour le déploiement automatique de composants*, in: *Journées Composants 2005*, Le Croisic, France, 2005, pp. 57–68.

[3] Bruneton, E., *Developing with Fractal*, France Telecom R&D (2004).
URL fractal.objectweb.org/

[4] Bruneton., E., T. Coupaye and J. Stefani, *Fractal Component Model Draft 2.0-2*, France Telecom R&D INRIA (2003).
URL fractal.objectweb.org/

[5] Distributed Management Task Force, *Common Information Model (CIM)*, CIM Infrastructure Specification, DMTF (2005).
URL www.dmtf.org/standards/cim

[6] Garlan, D., R. T. Monroe and D. Wile, *Acme: Architectural description of component-based systems*, in: G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, Cambridge University Press, New York, 2000 pp. 47–67.

[7] Gelfond, M. and V. Lifschitz, *The stable model semantics for logic programming*, in: *Proceedings of the 5th International Conference on Logic Programming* (1988), pp. 1070–1080.
URL homepages.inf.ed.ac.uk/dcspaul/publications/ggf12.pdf

[8] Hoareau, D. and Y. Mah o, *Constraint-Based Deployment of Distributed Components in a Dynamic Network*, in: *Architecture of Computing Systems (ARCS 2006)*, LNCS (2006).

[9] Hoffmann A. et al., *Specification of the deployment and configuration*, Deliverable D2.4, IST COACH Project (2003).

[10] Khare, R., M. Guntersdorfer, P. Oreizy, N. Medvidovic and R. Taylor, *xADL: Enabling Architecture-Centric Tool Integration with XML*, in: *Proc. of the 34th Hawaii International Conference on System Sciences, Volume 9* (2001), p. 9053.

[11] Kon, F., "Automatic Configuration of Component-Based Distributed Systems," Phd thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (2000).

[12] Medvidovic, N. and R. N. Taylor, *A classification and comparison framework for software architecture description languages*, Software Engineering **26** (2000), pp. 70–93.
URL citeseer.ist.psu.edu/medvidovic97classification.html

[13] OMG, *Deployment and Configuration of Component-based Distributed Applications*, Specification version 4, OMG (2006).
URL www.omg.org/cgi-bin/doc?formal/06-04-02

[14] Oquendo, F., *π-ADL: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures*, SIGSOFT Softw. Eng. Notes **29** (2004), pp. 1–14.

[15] Parrish, A. S., B. Dixon and D. Cordes, *A conceptual foundation for component-based software deployment.*, Journal of Systems and Software **57** (2001), pp. 193–200.

[16] R. Di Cosmo, *Report on Formal Management of Software Dependencies*, Delivrable WP2-D2.2, EDOS Project (2006).

[17] Reussner, R., I. Poernomo and H. Schmidt, *Contracts and quality attributes for software components*, in: W. Weck, J. Bosch and C. Szyperski, editors, *Proc. 8th Int'l Workshop on Component-Oriented Programming (WCOP'03)*, 2003.
URL citeseer.ist.psu.edu/703313.html

[18] Syrjänen, T., *A rule-based formal model for software configuration*, Research Report A55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland (1999).