Vers un déploiement sûr et flexible des composants logiciels

Meriem Belguidoum, Fabien Dagnat

Vers un déploiement sûr et flexible des composants logiciels

Meriem Belguidoum* , Fabien Dagnat**
* Université Mentouri
Département Informatique
Constantine, Algerie
**Télécom Bretagne
Brest, France
m.belguidoum@gmail.com, fabien.dagnat@telecom-bretagne.fr

ABSTRACT

Les applications deviennent de plus en plus complexes. Pour faciliter leur gestion elles sont représentées par des collections de composants qui sont partagés. Ainsi, une opération de déploiement concernant une application aura un effet sur toutes les applications qui l'utilisent. Une telle opération nécessite la connaissance préalable de toutes les *dépendances* de ces applications. Les approches de gestion du déploiement actuelles sont ad hoc et ne permettent pas de gérer correctement ces dépendances. Il est donc crutial de *maîtriser* et de *vérifier* le bon déroulement du déploiement. Pour cela, nous proposons un méta-modèle générique pour le déploiement automatique et *flexible* des composants ainsi qu'une description formelle de ce modèle pour vérifier et *garantir* le bon déroulement du déploiement. Enfin, nous proposons un système formel intégrant les propriétés *non fonctionnelles* dans la gestion du déploiement. L'intérêt est de pouvoir personnaliser le déploiement en fonction des exigences et des besoins pour le rendre plus flexible.

Keywords

composants logiciels, déploiement sûr, gestion des dépendances, modélisation, preuve formelle

1. INTRODUCTION

Le déploiement de logiciels est caractérisé par un ensemble de tâches qui couvre toutes les actions qui composent son cycle de vie depuis son développement jusqu'à son utilisation [8]. Ainsi, il ne s'agit pas uniquement d'installer ces logiciels, mais de les gérer jusqu'à leur désinstallation en passant par la mise à jour.

Les logiciels doivent fournir un spectre de plus en plus large de fonctionnalités et doivent rester réalisables à des coûts raisonnables. Pour cela, il doivent réutiliser des bouts de logiciels développés par ailleurs. Ainsi, un logiciel est devenu un ensemble de composants (sur une machine) partagés entre plusieurs logiciels. Lors d'un tel partage, on parle de *dépendances* [23] pour décrire aussi bien les exigences (en terme de services ou de ressources) d'un composant que l'effet qu'il a sur le système hôte (services fournis, services interdits, consommation de ressources, etc.).

Le résultat de ce partage est que toute opération de déploiement réalisée sur un logiciel peut avoir un impact (négatif) sur les autres logiciels du système hôte qui l'utilisent. Pour éviter de rendre inutilisable tout un ensemble de logiciels, chaque opération de déploiement doit s'assurer que le logiciel en cours de manipulation (1) spécifie explicitement ses dépendances, (2) que ses exigences sont satisfaites et (3) que l'effet de l'opération sur les autres logiciels de la machine hôte ne causera pas de problèmes. Ainsi, tout bon système de déploiement se doit de tenir compte des dépendances lors de ses opérations. Pour cela, il va devoir décrire explicitement ces dépendances et mettre en application un ensemble de règles pour les vérifier formellement. Pour offrir de réelles garanties, il conviendrait que la correction d'un tel ensemble de règles ait été prouvé. Or, la plupart des systèmes de déploiement actuels ne fournissent pas explicitement ces règles et ne prouvent pas leur correction.

La deuxième difficulté rencontrée pour le déploiement est la flexibilité. En effet, chaque type de plate-forme hôte dispose de son propre outil de déploiement. Par exemple, dans l'environnement Debian de Linux on utilise le gestionnaire de paquet apt, dans Windows on utilise Windows Installer pour la gestion des paquets msi, etc. Nous pouvons constater que ces outils étant fortement couplés au système d'exploitation hôte, les paquets créés pour l'un ne sont pas utilisables pour les autres. De plus, au dessus de ces plates-formes, ce sont créés des systèmes de déploiement ad-hoc spécialisés presque pour chaque application (extension d'eclipse, de firefox, de latex, etc). Pour diminuer cette diversité et rendre les systèmes plus génériques, il conviendrait de disposer d'un modèle commun de la notion de dépendances.

Enfin, la dernière difficulté est la nécessité de prise en compte des besoins non fonctionnels. Chaque composant fournissant des services peut en effet le faire avec des caractéristiques radicalement différentes (un niveau de sécurité différent par exemple) et doit également pouvoir spécifier ses préférences sur les services qu'il exige et leurs fournisseurs. Ainsi, les dépendances doivent également permettre de spécifier les propriétés non fonctionnelles des services exigées par un composant mais également les caractéristiques non fonctionnelles des services qu'il fournit. De plus, comme la liste de ces propriétés semble difficile à produire, il conviendrait de proposer un système de déploiement qui puisse être facilement étendu à de nouvelles propriétés. Pour cela, il faut proposer des règles paramétrées par ces propriétés.

Cet article a pour objectif de mettre l'accent sur la démarche suivie pour concevoir une infrastructure de déploiement sûre (pour garantir le bon déroulement du déploiement) et flexible (qui permet d'adopter une solution à la fois généri-

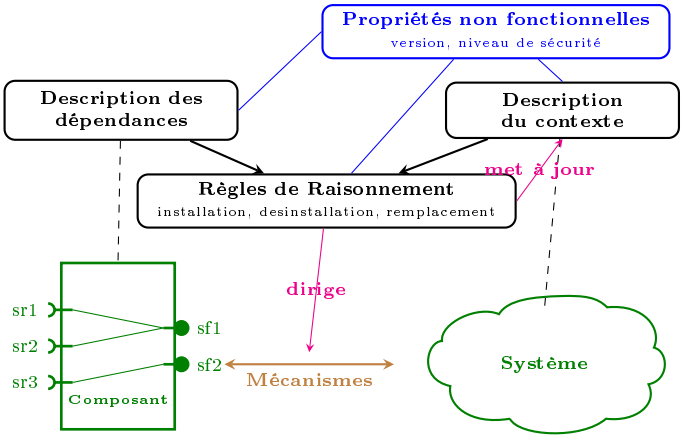

Figure 1: Principe du déploiement

que et personnalisable). Pour cela, nous avons procédé par trois étapes. La première étape représente la *modélisation générique* des concepts les plus intéressants du déploiement. La deuxième représente la *formalisation mathématique* du modèle proposé. Enfin, la troisième représente la *preuve* du bon déroulement du déploiement. Dans ce papier, nous allons nous concentrer plus sur les deux premières étapes. L'étape de la preuve ainsi que l'extension du système par les propriétés non-fonctionnelles sont détaillées dans [2].

La figure 1 décrit le principe de notre déploiement. À partir des modèles réels du composant et du système cible nous allons les décrire formellement de façon plus abstraite. Par la suite, nous allons vérifier les deux descriptions formelles à l'aide d'un ensemble de règles de raisonnement. Une fois que le déploiement est vérifié, ces règles vont diriger les mécanismes pour effectuer le déploiement réel et mettre à jour le contexte du système cible avec l'effet du déploiement du composant.

L'article est organisé comme suit. La section 2 présente une comparaison avec quelques travaux représentatifs existants. La section 3 présente notre modèle de déploiement générique avec ses différentes entités. Dans la section 4, une formalisation mathématique du modèle générique est décrite. Cette représentation recouvre la description formelle des entités de déploiement et la vérification des règles d'installation, de désinstallation et de mise à jour. La section 5 présente une description brève des propriétés de *sûreté* et de *réussite* des opérations de déploiement en général. La preuve de ces propriétés est décrite en détail dans [2]. Dans la section 6 nous décrivons une extension de la formalisation par la description des propriétés non-fonctionnelles. La section 7 conclut cet article en présentant quelques perspectives. Enfin, une annexe contient les règles d'installation, les règles de génération du graphe de dépendances et un exemple de vérification d'installation.

2. TRAVAUX CONNEXES

Les travaux présentés dans cet article concernent trois problématiques : la modélisation générique du déploiement, la description des dépendances et la vérification du déploiement. Nous pouvons, par conséquent, les comparer à des travaux menés selon ces trois axes.

2.1 Modélisation du déploiement

Les principaux travaux autour de la modélisation du déploiement suivent l'approche de modélisation proposée par l'ingénierie dirigée par les modèles. C'est ainsi qu'ils définissent des méta-modèles du domaine du déploiement ainsi que des outils reposant sur ces méta-modèles capable de déployer des logiciels. Pour cela, il faut que le développeur produise un modèle du déploiement de son logiciel décrivant ses propriétés et ses exigences.

Parmi ces travaux, nous pouvons citer la spécification D&C [17] de l'OMG, CADeComp [1] ainsi que ORYA [14, 16]. D&C est issue des travaux autour du modèle de composant CCM et propose un standard pour le déploiement et la configuration d'application distribuées à base de composants. CADeComp est une extension de cette spécification réalisée pour y intégrer la sensibilité au contexte et ainsi permettre à un déploiement de s'adapter à son contexte de réalisation. Enfin, ORYA est une plate-forme de déploiement développée au sein du laboratoire LSR de Grenoble qui a une approche plus globale. En effet, le déploiement est réalisé par des procédés et des fédérations, les procédés permettent de structurer et d'automatiser les différentes étapes et les fédérations permettent de lier les concepts des différents domaines.

Ces approches ont en commun la proposition d'un méta-modèle du déploiement sans se préoccuper de la vérification formelle du déploiement. À nos yeux cela représente une limite de ces modèles. Nous adoptons le même principe que ces approches pour proposer un modèle générique basé sur UML mais en l'enrichissant par une formalisation mathématique afin de pouvoir prouver sa correction par la suite. Ce modèle formel permet donc de (1) vérifier et donc garantir le bon déroulement d'une opération de déploiement et (2) fournir des règles de raisonnement indépendantes de la réalisation.

2.2 Description des dépendances

Au niveau de la description des dépendances, nous présentons une comparaison de notre approche de description avec celle utilisée dans le gestionnaire de paquets `apt` [21] de Debian et celle des modèles de composants orientés services [10].

2.2.1 *Les gestionnaires de paquets*

Dépendances / Descriptions		Debian
Dépendances obligatoires	**exigences**	`Depends : libc6 ..` `Installed-size : 1380`
	fournis	`Provides : mta`
Dépendances optionnelles	**exigences**	`Recommends: mail-reader` `Suggests: procmail, ..`
	fournis	-
Dépendances conflictuelles		`Conflicts: mta`

Table 1: La description des paquets Debian

Le tableau 1 décrit la description des dépendances d'un paquet Debian. Cette description ne permet pas d'exprimer qu'elle est la fonctionnalité supplémentaire (optionnelle) qu'un paquet peut fournir au système quand une certaine exigence est vérifiée. De plus, les gestionnaires de paquets ne

permettent pas de gérer plusieurs instances d'un même paquet ou plusieurs instances d'une même fonctionnalité d'un paquet. Cependant, avec notre description que nous allons présenter dans la section 4.1.1, nous pouvons exprimer qu'un service est fourni de façon optionnelle et qu'un composant fournit une ou plusieurs instances d'un même service. En effet, nous pouvons imaginer qu'un composant puisse avoir deux implantations différentes, par exemple, une implantation avec la meilleure qualité de service et une autre en mode dégradé et selon les exigences disponibles nous allons fournir l'un ou l'autre.

Un autre point de comparaison est la structuration des paquets et des composants. Dans la structuration des paquets de Debian, il existe une séparation entre les différents paquets, par exemple, les paquets contenant les binaires (`bin`) sont séparés de ceux contenant des sources (`src`), etc. Ainsi, la dépendance entre ces différents paquets peut être complexe à gérer. Notre approche, vise à encapsuler les différents paquets avec leurs dépendances dans un seul bloc et de ne fournir à la demande que les paquets appropriés selon les besoins de l'utilisateur et les exigences du système. L'idée est de simplifier les dépendances entre les différents paquets ainsi que leurs dépôts. L'originalité de notre contribution est au niveau du langage proposé qui permet d'exprimer les exigences de chaque fonctionnalité qu'un paquet peut fournir, chose qui n'existe pas dans les composants par exemple. En effet, toutes les exigences et les fonctionnalités du composant sont exprimées sans aucune précision sur la relation qui existe entre chaque fonctionnalité et ses exigences qu'un paquet peut fournir. Nous avons donc plus de flexibilité dans l'utilisation et une gestion d'un dépôt de paquets plus simple.

2.2.2 *Les composants orientés services*

Le *service binder* [10] est un composant orienté services qui manipule des instances de composants avec des propriétés qui peuvent varier. Cependant, ces propriétés sont propres au composant et pas aux services qu'il fournit. Dans notre approche, nous pouvons avoir le même service avec plusieurs propriétés. De plus, les composants sont considérés comme des boites noires car nous n'avons aucune idée sur la relation entre leurs exigences et ce qu'ils fournissent. La description d'un composant contient l'ensemble de ses exigences et l'ensembles de ses services fournis mais pas la description explicite des dépendances entre chaque service fourni et ses exigences (voir le composant représenté à gauche de la figure 2). Nous visons donc à avoir plus de transparence au niveau de la description des dépendances à l'intérieur de chaque composant (voir le composant situé à droite de la figure 2) pour pouvoir contrôler la fourniture des services en fonction des exigences et des besoins. Cette approche peut être rapprochée de la notion de contrat paramétré proposé par [19] pour permettre la spécification précise de la qualité des services fournit par un composant.

2.3 Vérification du déploiement

Peu de travaux aborde la problématique de la correction du déploiement, on peut citer les travaux menés dans le cadre du projet EDOS [13, 12] qui propose un *framework* pour la spécification formelle du déploiement des paquets et pour la résolution du problème d'installabilité pour les distributions Linux ainsi que sat4J [20] qui est un outil basé sur la saturation de contraintes utilisé pour vérifier les dépendances des extensions de Eclipse. Ces deux travaux utilise la même approche à savoir, il propose une méthode d'encodage des dépendances de déploiement en une formule logique qui est résolu par un outil SAT.

Figure 2: La description d'un composant vs. notre approche

Nous adoptons le même principe, mais avec un formalisme basé sur la logique en utilisant un langage de description de dépendances plus expressif. De plus, dans notre approche, nous nous intéressons pas la recherche d'une solution pour une équation booléenne mais plutôt à la démontrer si elle est vraie à l'aide d'un ensemble de règles logiques. Ainsi, dans cas où le déploiement est possible, son *bon déroulement* est *prouvé* en même temps que sa réalisation.

3. MODÉLISATION DU DÉPLOIEMENT

Pour l'automatisation du déploiement, nous avons besoin de définir et de construire des modèles abstraits. L'intérêt de ces modèles est double, ils permettent d'avoir d'un coté un aspect paramétrable et extensible donc plus de flexibilité au niveau du système et de l'autre coté un raisonnement indépendant des contraintes de réalisation.

3.1 Les entités du déploiement

Les entités du déploiement que nous manipulons sont les composants logiciels et les services. Nous adoptons la définition de [24]. Dans cette définition, un composant est considéré comme une boite noire représentant une entité indivisible encapsulée par ses interfaces. Elle implique donc l'existence des notions de composition, d'interface, de dépendance et de cycle de vie indépendamment des contraintes sur le type d'interaction avec d'autres composants.

Les composants requièrent et fournissent des services, le service représente donc l'unité logiciel qu'un composant peut fournir ou exiger. Le descripteur d'un composant est un descripteur des dépendances qui existent entre ses services fournis et ses services requis. La réalisation et la vérification de l'installation et de la mise à jour des composants sont basés principalement sur ce descripteur de dépendances.

Au niveau du déploiement, le plus important est la représentation des dépendances (relation entre les interfaces requises et fournies). Pour cela, nous avons choisi de représenter le composant par ses dépendances, ses services requis, ses services fournis et ses propriétés. La figure 3 représente le métamodèle de composant que nous proposons. Un composant (*composite*) peut être composé de plusieurs composants. La description usuelle d'une dépendance est représentée par la relation d'*InterDépendanceComposant* qui décrit la liaison entre deux composants différents. Le lien entre un service requis par un composant et un service fourni par un autre composant représente la relation d'*InterDépendanceService* (le service requis et le service fourni doivent être compatibles). Cette relation est définie par l'assembleur de l'appli-

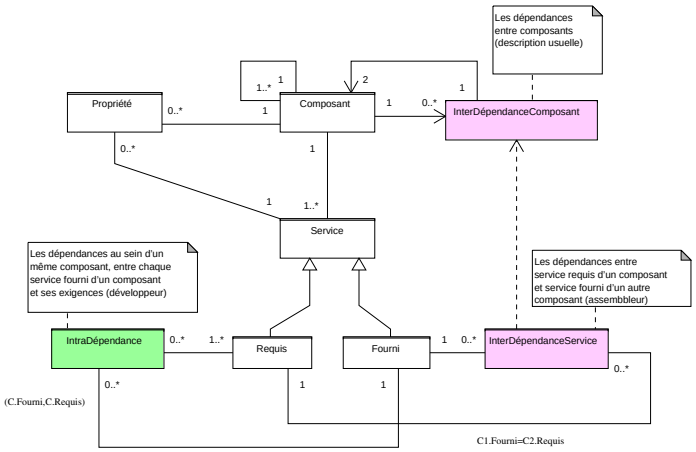

Figure 3: Le méta-modèle d'un composant

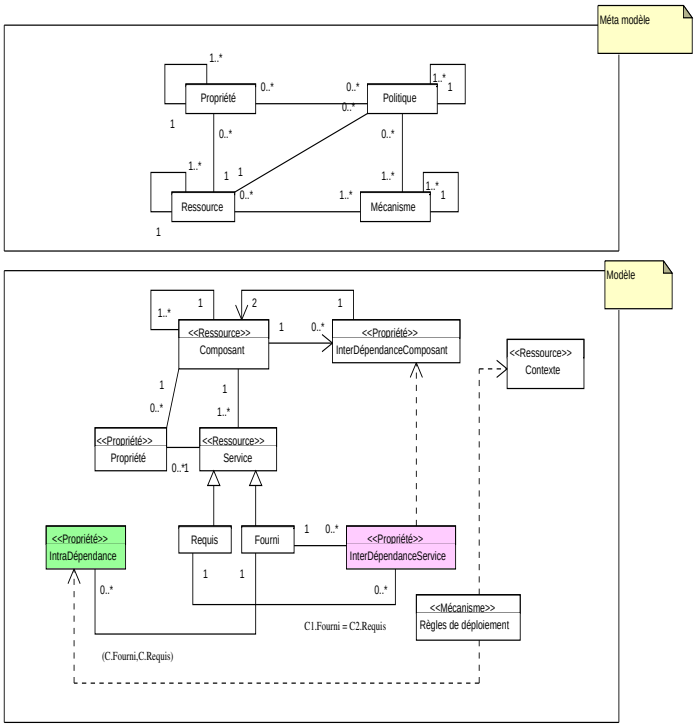

Figure 4: Le méta-modèle du déploiement

cation, elle est un raffinement de la relation *InterDépendanceComposant* qui a la même signification mais décrite en termes de composants. Enfin, la relation *IntraDépendance* est la partie la plus originale de notre contribution, elle permet d'exprimer le lien entre chaque service fourni par un composant et l'ensemble de ses exigences. L'intra-dépendance est définie par le concepteur ou le développeur de l'application. Étant donné que ces intra-dépendances sont explicites dans notre modèle, le composant n'est donc plus considéré comme une boite noire.

3.2 Un modele générique du déploiement

Nous constatons qu'avec l'évolution des systèmes informatiques le besoin de gestionnaires de déploiement génériques s'accroît de plus en plus. En effet, malgré tout les progrès dans le domaine du déploiement, le besoin de généricité et flagrant. Comme nous visons une infrastructure de déploiement flexible et adaptable, nous proposons un méta-modèle qui regroupe les concepts les plus pertinents du déploiement. Ce modèle représente la brique de base sur laquelle nous nous basons pour décrire et vérifier formellement les différentes phases de déploiement.

La figure 4 illustre notre méta-modèle générique avec une spécialisation (au dessous) que nous avons adoptée.

Dans le premier niveau (le modèle générique) nous défendons un découpage en quatre modèles [3] : les ressources, les mécanismes, les politiques et les propriétés. Dans ce découpage, les concepts sont classés suivant leur *rôle* durant le déploiement. Ainsi, les **ressources** sont manipulées durant le déploiement (n'importe quelle entité intervenant au niveau du déploiement). Par exemple, lors de l'installation d'une *application* sur une *machine*, l'application et la machine sont des ressources. Les **mécanismes** décrivent la façon dont sont réalisées les opérations agissant sur les ressources (installer, copier, supprimer, rajouter, etc.). Les **politiques** décrivent les choix des mécanismes à appliquer en fonction de l'état des ressources. Par exemple, une politique de *sécurité* peut interdire l'installation des programmes qui sont susceptibles de diminuer le niveau de sécurité du système cible (par exemple, l'interdiction de l'installation d'un composant `ftp`). Enfin, les **propriétés** permettent de décrire les caractéristiques fonctionnelles des ressources : par exemple, l'entité (`E.version=1.0.2`) ou le système cible (`RAM=64Mo`), ou les caractéristiques non fonctionnelles (`E.security=high`) ou un autre niveau de propriétés qui exprime les buts à atteindre par le système (par exemple, sûreté, succès). Les politiques portent souvent sur des propriétés du système cible ou de l'entité.

Ces modèles sont récursifs, c'est-à-dire qu'une entité manipulée (ressource, mécanisme, politique ou propriété) est également composée de sous-entités. Par exemple, une ressource de type `machine` contient des ressources de type `logiciel` qui contiennent des ressources de type `composant` qui sont composées de ressources de type `service`. Ainsi, chaque modèle fournira une description hiérarchique et structurée.

Le modèle de la figure 4 est une spécialisation du niveau au-dessus. Nous avons divisé ce modèle en deux parties. La première partie est liée aux composants et au contexte du site cible qui sont des spécialisations des ressources. Le composant est décrit avec ses différentes relations de dépendances dans la section 3.1. Le *contexte* cible représente toutes les entités qui constituent les différentes ressources. Elle est une abstraction de la structure arborescente des différents éléments constituant le site cible. Par exemple, un réseau est constitué de plusieurs machines qui ont chacune un OS et plusieurs applications qui sont à leur tour composées de plusieurs composants, etc. La deuxième partie du méta-modèle, représente le moteur de raisonnement qui est le cœur de notre travail et qui représente une spécialisation des mécanismes. Cette partie est réalisée par un ensemble de *règles de déploiement* qui utilisent la description des ressources (les dépendances des composants et la description du site cible) pour vérifier et réaliser le déploiement. Ces règles peuvent également utiliser les propriétés et les

politiques des ressources pour effectuer le déploiement.

4. FORMALISATION MATHÉMATIQUE DU MODÈLE GÉNÉRIQUE

Une méthode formelle se bâtit à partir de deux ingrédients principaux : un langage de *spécification* et un système de *vérification*. Pour la description et la réalisation du déploiement, nous avons adopté cette approche formelle. Son intérêt est d'avoir des spécifications munies d'une sémantique sûre et surtout de pouvoir *prouver* les propriétés attendues du système de déploiement. Dans cette section, nous présentons une formalisation du modèle générique proposé. Pour cela, nous commençons par la description formelle des entités de déploiement ensuite la description des règles de vérification du déploiement. Nous nous sommes concentrés sur les opérations d'installation, de désinstallation et de mise à jour.

4.1 Description des entités du déploiement

L'étude des dépendances des différentes applications est primordiale dans le déploiement. Une dépendance d'une application représente la relation entre les fonctionnalités qu'elle peut offrir et ses exigences. L'analyse des dépendances représente la résolution de l'ensemble des contraintes de déploiement liées à l'application qui va être déployée par rapport à l'existant (les contraintes liées au système cible) et aux exigences de l'utilisateur.

4.1.1 Description des dépendances des composants

Les dépendances peuvent être vues comme un contrat représentant un ensemble de *pré* et *post-conditions*. Une classification des contrats est présentée dans [6].

La démarche que nous avons adoptée est de déterminer les différents types de dépendances liées au déploiement de composants. Par la suite, il faut pouvoir les décrire dans un langage formel pour pouvoir les vérifier dans le contexte de l'environnement afin de garantir le bon déroulement du déploiement. Les dépendances permettent de disposer d'une bonne description des exigences et des effets des applications. Une telle description va permettre au système de déterminer les cas de conflits et d'incompatibilité, de vérifier les conditions de déploiement et de prévoir les traitements appropriés pour des cas complexes par exemple.

Avant de donner les définitions de nos descriptions, nous présentons un exemple pour en donner l'intuition de notre description des dépendances. Cet exemple est un serveur mail dans un système Linux, il est présenté dans la figure 5. Le serveur mail que nous présentons est constitué des composants suivants : `POSTFIX`, `PROCMAIL`, `FETCHMAIL` et `THUNDERBIRD`. Ces composants sont décrits ci-dessous :

- `POSTFIX` est un serveur SMTP pour le relais de mails. Il joue le rôle d'un *Mail Transport Agent* (`MTA`).
- `FETCHMAIL` permet de récupérer du mail par un protocole de transport de courrier électronique (Pop ou IMAP), d'un hôte distant vers une machine locale (les messages sont redirigés vers la messagerie locale).
- `PROCMAIL` gère les mails reçus, il permet par exemple de filtrer les mails, de faire des redirections en fonction de l'émetteur, du sujet, de la taille du mail, etc. Il traite la messagerie locale. Il joue le rôle d'un MDA *Mail Deliver Agent*

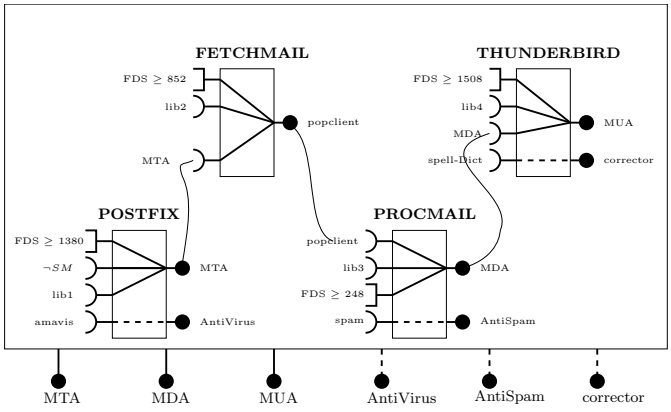

Figure 5: Un exemple de serveur mail

- `THUNDERBIRD` est un gestionnaire de courrier électronique qui permet de lire et composer des mails (MUA : *Mail User Agent*)

Les services fournis sont représentés par des ronds noirs. Les exigences peuvent être logicielles et correspondre à des services représentés par des demis-cercle (*lib*1, *lib*2, *amavis*, etc.) ou portées sur l'environnement. Ces dernières seront alors de la forme une variable comparée à une valeur et sont représentées par des crochets. Ainsi, par exemple, une taille d'espace disque disponible est exigée (`Free disk space` (FDS) $\geq$ 852ko).

nous présentons la définition précise de la relation entre les services fournis et les services requis par le même composant (*intra-dépendance*) ou deux composants différents (*interdépendance*). L'exemple du serveur mail de la figure 5 illustre ces dépendances au niveau d'un même composant et au niveau de plusieurs composants[1]. Il existe trois formes principales de dépendances :

- une dépendance *obligatoire* (représentée à l'intérieur d'un composant par une ligne continue) représente un besoin nécessaire d'un composant pour qu'il puisse être installable et utilisable. Par exemple, un serveur mail a besoin d'un système d'exploitation particulier, d'un espace disque libre suffisant, de certaines librairies, etc.
- une dépendance *optionnelle* (représentée par un trait discontinu) spécifie qu'un composant peut fournir un service optionnel. Le composant peut donc être installé même si le service optionnel n'est pas fourni (dans le cas où ses exigences ne sont pas disponibles). Par exemple, `POSTFIX` peut fournir un anti-virus si le composant `AMAVIS` est disponible. Dans le cas contraire, `POSTFIX` peut toujours être installé et fournit le service de transport de mail `MTA` sans un anti-virus `av`.
- une dépendance *négative* (exprimée par une négation) exprime les conflits entre composants et services. Par exemple, `POSTFIX` ne peut être installé si un autre `MTA` est déjà installé (tel que `SENDMAIL` par exemple).

Une intra-dépendance représente la relation entre ce qui est requis et ce qui est fourni au niveau d'un même composant.

[1]Les dépendances sont simplifiées par rapport aux dépendances réelles.

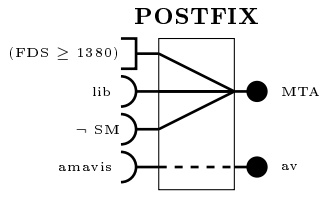

Figure 6: Les intra-dépendances du composant `POSTFIX`

Les besoins du composant représentent ses *pré-conditions* et les services qu'il fournit représentent ses *post-conditions*. Par exemple, la figure 6 représente les *intra-dépendances* du composant `POSTFIX`, pour chacun de ses services fournis (*post-conditions*), il exige un certain nombre de *pré-conditions*. Pour fournir le service `MTA`, `POSTFIX` exige un minimum d'espace de disque libre (`FDS`$\geq$ 1380), une librairie (`lib`) et interdit le composant `SM` (sendmail). Le service anti-virus `av` est fourni de manière optionnelle selon la disponibilité du service `amavis`.

Le langage de description des intra-dépendances se base sur la logique des prédicats. Les pré-conditions sont décrites par des prédicats et la dépendance représente la description des pré-conditions et leurs post-conditions. Le langage de dépendance est défini par la grammaire ci-dessous, s représente le nom du service et c le nom du composant :

$$D ::= P \Rightarrow s \mid D \bullet D \mid D \# D \mid ?\, D$$

$$P ::= true \mid P \wedge P \mid Q$$

$$Q ::= Q \vee Q \mid [v\ O\ val] \mid \neg\, s \mid \neg\, c \mid c.s \mid s$$

$$O ::= > \mid \geq \mid < \mid \leq \mid = \mid \neq$$

Les dépendances peuvent être simples ($P \Rightarrow s$) ou composées ($D \bullet D$, $D \# D$, $?\, D$). Les détails sur la grammaire des dépendances sont présentés ci-dessous :

- $P \Rightarrow s$ est la dépendance la plus simple, elle est représentée par une suite de pré-conditions décrites par le prédicat P et une post-condition qui représente la fourniture du service s;
- $D \bullet D$ est la conjonction de dépendances D_1 et D_2;
- $D \# D$ est la disjonction de dépendances D_1 et D_2;
- $?\, D$ est la dépendance D rendue optionnelle.

Au niveau des prédicats, le langage est un sous ensemble de la logique des prédicats sans l'ensemble des fonctions. Le prédicat *true* désigne une dépendance toujours vérifiée et le service en partie droite de l'implication est toujours fourni. Le prédicat doit être défini en forme normale conjonctive (composée de conjonction de disjonction) pour faciliter l'écriture de certaines règles.

Le couple $[v\ O\ val]$ représente l'ensemble des valeurs val données pour une certaine variable v, en sachant que O représente un opérateur de comparaison ($>, \geq, <, \leq, =$,ou $\neq$). Les prédicats $\neg\, s$ et $\neg\, c$ permettent de préciser que le service s respectivement, le composant c, sont interdits. Enfin, le prédicat s exige la disponibilité du service s de n'importe quel composant, par contre $c.s$ exige le service s fourni par le composant c.

Par exemple, les principales dépendances du composant `POSTFIX` représentées graphiquement dans la figure 6 sont : l'espace disque disponible ($FDS \geq 1380$), les librairies (`lib`), l'absence de `SENDMAIL` (SM), le service `amavis` pour fournir l'anti-virus `av` est optionnel. L'expression des dépendances de `POSTFIX` selon la figure 6 est:

$$([FDS \geq 1380] \wedge (SM) \wedge lib \Rightarrow MTA) \bullet ?\, (amavis \Rightarrow av)$$

4.1.2 Description d'un assemblage de composants

De même que la notion de composant, la notion d'architecture est importante car elle permet d'organiser et d'assembler les composants lors de la construction d'une application. Cette architecture est généralement décrite en utilisant un ADL (*Architecture Description Language*) [22] qui définit l'ensemble des artefacts (composant, connecteur, liaison, port, interface, etc.) et une syntaxe concrète qui permettent aux développeurs de décrire la structure des applications.

Un assemblage produit un nouveau composant à partir d'un ensemble de composants. Dans le cadre notre travail, nous n'allons pas définir un assemblage en connectant précisément une exigence d'un composant au service fourni d'un autre composant, mais seulement en spécifiant l'ensemble des composants de l'assemblage. Notre outil calcule alors l'ensemble des services que peut fournir l'assemblage et ses exigences (sa dépendance *interne*). Pour cela, il va déterminer toutes les connections possibles entre sous-composants. Par exemple, si on assemble un composant C_1 de dépendance $s_1 \Rightarrow s_2$ et un composant C_2 de dépendance $s_2 \Rightarrow s_3$, le service s_2 de C_1 sera utilisé pour satisfaire l'exigence s_2 de C_2. L'assemblage a donc la dépendance interne $s_1 \Rightarrow \{s_2, s_3\}$. Cette opération d'assemblage s'applique à un ensemble de composants avec leurs dépendances.

4.1.3 Description du systŁme cible

Les ressources et l'architecture du système cible sont représentées par le contexte. Ce dernier pourrait être représenté par l'union des dépendances de tous ses composants. Mais, le calcul et la manipulation de cette union sont plutôt difficiles à gérer. Nous choisissons donc d'utiliser une approximation de cette union. Pour assurer un déploiement sûr, cette approximation doit être sûre et contenir les services disponibles du système cible, les composants qui les fournissent, ainsi que l'ensemble des exigences et des conflits du système. Nous avons également besoin des valeurs des variables d'environnement.

Le *Contexte* est composé de (1) l'ensemble des valeurs des variables d'environnement noté $\mathcal{E}$ (2) l'ensemble $\mathcal{C}$ des quadruplets $(c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ mémorisant, pour chaque composant installé c, l'ensemble de ses services fournis $\mathcal{P}_s$, celui de ses services interdits $\mathcal{F}_s$ et celui de ses composants interdits $\mathcal{F}_c$ et (3) un graphe de dépendance $\mathcal{G}$ stockant l'ensemble des dépendances existantes dans le système.

Un nœud du graphe $\mathcal{G}$ est un service disponible avec son fournisseur ($c.s$). Un arc est une paire de nœuds $n_1 \mapsto n_2$ où n_2 dépend de n_1. Chaque arc est étiqueté par le type de dépendance entre les deux nœuds qui soit obligatoire M (*Mandatory*) soit optionnel O (*Optional*). Le graphe de dépendance représente donc la configuration actuelle du système cible et son architecture en termes de composants et services. Il est construit pendant l'installation et utilisé

pendant la désinstallation. Il est également mis à jour par chaque opération du déploiement.

La figure 7 représente une partie du graphe de dépendance d'un système qui contient le serveur mail présenté dans la figure 5. Le composant `POSTFIX` fournit un service obligatoire `MTA` qui requiert la librairie `lib` fournie par le composant C_1 (C_1`.lib`) et un service optionnel anti-virus `av` qui requiert le service `amavis` fourni par le composant `AMAVIS` (`AMAVIS.amavis`). La dépendance obligatoire est représentée donc par un arc continu ($\mapsto$) qui désigne l'arc M (entre C_1`.lib` et `POSTFIX.MTA`) et la dépendance optionnelle par un arc discontinu qui désigne l'arc O (entre `AMAVIS.amavis` et `POSTFIX.av`).

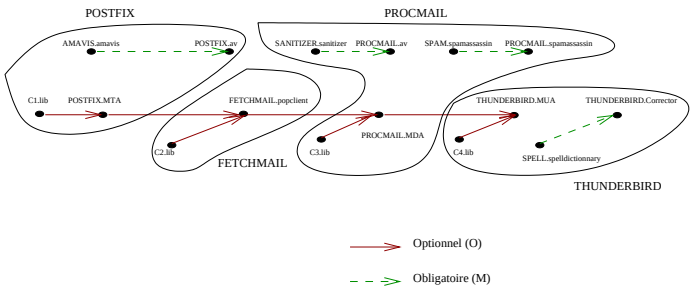

Figure 7: Une partie du graphe de dépendance du serveur mail de la figure 5

Pour simplifier la présentation de nos règles de déploiement, nous définissons des fonctions qui calculent l'ensemble des services disponibles AS, celui des composants disponibles AC, celui des services interdits FS et enfin celui des composants interdits FC. Ces fonctions sont décrites dans l'annexe.

4.2 Vérification formelle du déploiement

Pour déduire la possibilité d'un déploiement (installation, désinstallation, mise à jour), nous nous sommes basés sur le calcul logique. Un calcul en logique est un ensemble de règles permettant en un nombre fini d'étapes, selon des règles explicites et selon des procédés mécanisables de déterminer si une proposition complexe est vraie ou fausse. D'une façon générale, les règles exposées dans nos travaux sont celles de la déduction naturelle pour le calcul des propositions. La recherche d'une déduction logique consiste à analyser les prémisses, c'est-à-dire à les démontrer, et les rassembler pour faire des formules que l'on peut enchaîner logiquement jusqu'à la conclusion.

Les déductions correctes sont toutes celles qui respectent rigoureusement les règles fondamentales. Ce type de règles est appelé règle d'inférence, par exemple dans la règle suivante :

$$\textsc{Règle-Inférence:}\ \frac{S \vdash P_1 \ldots S \vdash P_n}{S \vdash C}$$

Le principe est de vérifier les *prémisses*, dans ce cas P_1 jusqu'à P_n. Si ces *prémisses* sont vérifiées alors la règle est déclenchée et la *conclusion* C est démontrée.

4.2.1 Vérification de l'installation

L'installation d'un composant dans un système correspond à l'ajout d'un composant dans un environnement. Pour cela, nous procédons en deux étapes :

- Vérification de l'*installabilité* du composant, en répondant à la question est ce que le composant peut être installé dans le système ? Pour répondre à cette question, il faut appliquer un ensemble de règles qui vérifient les exigences du composant par rapport à l'environnement. De plus, ces règles permettent de vérifier en amont l'impact de l'installation de ce composant sur le système et ainsi de s'assurer qu'il ne perturbera pas le système (par exemple, il ne sera pas en conflit avec les autres composants).

- La deuxième étape consiste à calculer l'*effet* de l'installation du composant sur le système en mettant à jour l'ensemble des services et des composants disponibles, l'ensemble des services et des composants interdits, ainsi que le graphe de dépendance (cf. description du système cible section 4.1.3).

Installabilité

Avant d'autoriser l'installation d'un composant, il faut s'assurer que (1) le composant n'est pas interdit par le système cible, (2) les services qu'il requiert sont disponibles dans le contexte et (3) il ne fournit pas des services interdits par le système cible. Plus formellement, nous décrivons l'étape d'installabilité d'un composant comme suit :

Définition 1 ***(Installabilité)*** *Un composant c avec une dépendance D est* installable *dans un contexte Ctx ssi le composant n'est pas interdit et la dépendance D est vérifiée par les règles de vérification de l'installabilité. Ces règles sont présentées dans la figure 8 :*

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

Les règles de vérification de l'installabilité de la figure 8 permettent de garantir que toutes les dépendances *obligatoires* du composant sont vérifiées. Pour une dépendance simple exprimée comme suit : $P \Rightarrow s$, cela veut dire que le prédicat P doit être vrai et le service s n'est pas interdit par le contexte du système cible (CTriv). L'évaluation du prédicat P dans le contexte Ctx est présentée dans la première partie de la figure (les règles $Ctx \vdash_P P$). Ces règles correspondent à l'évaluation de la grammaire des prédicats déjà présentée dans la section 4.1.1. La fonction $CalcF$ permet de déterminer les services et les composants qui sont interdits en collectant les négations introduites dans les prédicats de la dépendance. Les règles d'installabilité ainsi que la fonction $CalcF$ sont décrites en détail dans [4].

Installation

Une fois que l'installabilité du composant est prouvée, l'effet de l'installation de ce dernier dans le système cible doit être calculé. Cet effet concerne les nouveaux services disponibles, les nouveaux services interdits, les nouveaux composants interdits ainsi que les nouvelles dépendances (représentées par un graphe de dépendance). Nous allons présenter dans un premier temps avant de décrire les règles d'installation, les fonctions qui permettent de calculer l'effet de l'installation. Le graphe de dépendance est construit pendant la phase

Évaluation des prédicats :

$$\textsc{PTrue}: Ctx \vdash_P true \qquad \textsc{PAnd}: \frac{Ctx \vdash_P P_1 \qquad Ctx \vdash_P P_2}{Ctx \vdash_P P_1 \wedge P_2}$$

$$\textsc{POrL}: \frac{Ctx \vdash_P Q_1}{Ctx \vdash_P Q_1 \vee Q_2} \qquad \textsc{POrR}: \frac{Ctx \vdash_P Q_2}{Ctx \vdash_P Q_1 \vee Q_2}$$

$$\textsc{PVar}: \frac{Ctx.\mathcal{E}(v)\ O\ V}{Ctx \vdash_P [v\ O\ V]} \qquad \textsc{PNotS}: \frac{s \notin AS(Ctx)}{Ctx \vdash_P \neg s}$$

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

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

Règles d'installabilité :

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

$$\textsc{CAnd}: \frac{Ctx \vdash_C D_1 \qquad Ctx \vdash_C D_2}{Ctx \vdash_C D_1 \bullet D_2} \qquad \textsc{COpt}: Ctx \vdash_C ?\, D$$

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

Figure 8: Les règles d'installabilité et d'évaluation des prédicats

d'installation, il représente l'ensemble de toutes les dépendances des composants qui sont vérifiées dans le système (architecture du système).

Définition 2 ***(Calcule du Graphe)*** *Le graphe de dépendance $\mathcal{G}$ représente la relation entre les services requis et les services fournis du même composant ou des composants différents. Il représente donc les interdépendances et les intradépendances. Les règles utilisées pour la construction du graphe de dépendance sont présentées dans la figure 10 de l'annexe.*

Le principe des règles qui calculent le graphe de dépendance est de s'assurer que l'ensemble des exigences de chaque service fourni est disponible dans le contexte (cf. les prémisses des règles GServC et GServ de la figure 10). Une fois que ces services ($c'.s'$) sont disponibles dans le contexte les nœuds qui y correspondent sont liés avec celui correspondant au service fourni $c.s$ (cf. les conclusions des règles GServC et GServ). L'extrémité initiale de chaque arc représente donc un nœud requis par le nœud de son extrémité finale (par exemple, pour l'arc (a,b) : b dépend de a).

L'installation est définie comme suit :

Définition 3 ***(Installation)*** *L'installation d'un composant c de dépendance D dans un contexte Ctx a un effet sur : l'ensemble des services fournis $\mathcal{P}_s$, l'ensemble des services interdits $\mathcal{F}_s$, l'ensemble des composants interdits $\mathcal{F}_c$ et le graphe de dépendance $\mathcal{G}$. Ces effets sont calculées par les règles présentées dans la figure 11 de l'annexe.*

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

Au niveau de la prémisse de la règle IComp nous vérifions la dépendance D dans Ctx avec l'hypothèse que c est installable, nous calculons par la suite l'ensemble des services fournis $\mathcal{P}_s$, l'ensemble des services interdits $\mathcal{F}_s$, l'ensemble des composants interdits $\mathcal{F}_c$, le graphe de dépendance relatif à cette dépendance $\mathcal{G}_c$. Ce calcule sera propagé au niveau de la conclusion et le composant c sera avec l'effet calculé. Le nouveau contexte sera égal à l'ancien contexte avec en plus le composant c installé (avec son quadruplet) et le graphe résultant de sa dépendance vérifiée. Le nouveau graphe de dépendance résultant est :

$\mathcal{G}_c = \{n_1 \overline{\longmapsto} n_2 \mid n_1 \in AS(Ctx) \wedge n_2 \in c.\mathcal{P}_s\}$.

Une fois $D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}_c$ est vérifiée dans le contexte ($Ctx \vdash_I c : D \Rightarrow \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}_c$), le nouveau contexte est calculé comme suit :

$\boldsymbol{Installation(Ctx, c : D) =}$
$\boldsymbol{Ctx.\mathcal{E}, Ctx.\mathcal{C} \cup (c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c), Ctx.\mathcal{G} \cup \mathcal{G}_c}$ Les autres règles d'installation sont décrites en détail dans [4], la vérification de l'installation de `POSTFIX` est présenté dans l'annexe.

4.2.2 *Vérification de la désinstallation*

La désinstallation d'un composant est également réalisée en deux étapes :

- La première étape vérifie la faisabilité de la désinstallation en se basant sur le graphe de dépendance. Cette vérification consiste à s'assurer que les services fournis par le composant en question ne sont pas utilisés par d'autres composants.
- La deuxième étape consiste à calculer l'effet de la désinstallation sur le contexte, i.e. désinstallation effective du composant et de tous ses services fournis au niveau du contexte ainsi que l'ensemble de tous les services qui en dépendaient de façon optionnelle (directement ou indirectement) et qui ne possèdent pas d'autres exigences dans le graphe (ne représentent pas des extrémités finales des arcs du graphe). Les nœuds et les arcs correspondants dans le graphe seront supprimés.

Désinstallabilité

D'un point de vue formel, au niveau de la première étape, on utilise une règle de désinstallabilité qui s'assure qu'aucun service déjà utilisé par d'autres composants ne sera supprimé. La règle se base sur le graphe de dépendance du contexte. Un composant peut être désinstallé si aucun de ses services fournis n'est utilisé par d'autres composants de façon obligatoire. Pour cela, pour chaque service fourni, il faut s'assurer qu'il ne possède aucune dépendance obligatoire avec d'autres services. Ainsi, un service peut être désinstallé s'il n'est pas utilisé (i.e. il est une feuille du graphe de dépendance) ou il est utilisé de façon optionnelle (tous les chemins partant de ce service aux feuilles du graphe sont des arcs optionnels.

Désinstallation

Une désinstallation a un effet sur les composants (leurs services fournis, leurs services interdits, et leurs composants interdits) et sur le graphe de dépendance. L'ensemble des nœuds qui doivent être supprimés du graphe de dépendance contient tous les services fournis par c et tous les services optionnels qui en dépendent directement ou indirectement et qui ne représentent pas une extrémité finale des arcs du graphe. Une fois que la désinstallation est réalisée, Ctx est

mis à jour en supprimant c (et ses services fournis, services interdits et composants interdits) de $\mathcal{C}$ et en supprimant du graphe $\mathcal{G}$ tous les nœuds concernés par les services supprimés. Si l'on suppose que l'ensemble de ces nœuds est noté N, le graphe résultant de la désinstallation est calculé en supprimant ces nœuds avec leurs arcs associés dans la graphe. Il est calculé comme suit :

$$\mathcal{G} \setminus N = \{n_1 \overset{x}{\longmapsto} n_2 \mid n_1 \overset{x}{\longmapsto} n_2 \in \mathcal{G} \wedge n_1 \notin N \wedge n_2 \notin N\}$$

Le nouveau contexte est donc calculé comme suit :

$$\mathbf{Désinstallation}(\boldsymbol{Ctx}, \boldsymbol{c}) = \boldsymbol{Ctx.\mathcal{E}}, \boldsymbol{Ctx.\mathcal{C}} \setminus (\boldsymbol{c}, -, -, -), \boldsymbol{Ctx.\mathcal{G}} \setminus \boldsymbol{N}$$

Toutes les définitions formelles des règles de désinstallation sont décrites en détail avec un exemple de désinstallation dans [4].

4.2.3 Vérification de la mise à jour

La mise à jour des composants est une phase de déploiement très importante à gérer. Elle ne consiste pas seulement de désinstaller l'ancien composant et d'installer le nouveau. En effet, cette phase est fortement liée au problème de substituabilité qui est une propriété essentielle utilisée en particulier pour l'adaptation d'une application. Le problème de la substituabilité consiste à vérifier la possibilité de remplacer un composant par un autre. Des exemples sur différents cas de figures de substitution sont présentés pour illustrer notre approche de vérification. L'étude détaillée du problème de substitution est présentée dans [5], nous exposons dans ce papier une partie qui décrit le principe de la substitution.

Pour expliquer notre approche de substituabilité, nous commençons par des définitions des deux principales formes de substituabilité : la substituabilité stricte indépendamment du contexte et substituabilité contextuelle par rapport à un certain contexte. Les définitions que nous proposons sont les suivantes :

Définition 4 ***(Substituabilité stricte)***
Un composant C_{old} est substituable strictement *par un composant C_{new} ssi ils fournissent exactement les mêmes services et C_{new} exige moins ou autant de services que C_{old}.*

Définition 5 ***(Substituabilité contextuelle)***
Un composant C_{old} est substituable par un composant C_{new} dans un contexte Ctx ssi :

- *toutes les nouvelles exigences de C_{new} sont vérifiées dans Ctx.*
- *aucun des nouveaux services fournis (s'il y en a) n'est en conflit avec Ctx.*
- *aucun des services fournis par C_{old} et pas fournis par C_{new} n'est nécessairement utilisé dans Ctx.*

Les substituabilité stricte et contextuelle sont abordées dans [7] mais de façon différente de la notre. Il considère que le nouveau composant doit fournir au moins la même chose et exige au plus la même chose (généralisation des besoins et spécialisation des offres). Cependant, dans notre cas, les services fournis en plus ne doivent pas être en conflit avec d'autres services du contexte. Ainsi, dans [7], il n'y pas de vérification des nouvelles exigences ou des nouveaux services fournis s'ils existent. En effet, de notre point de vue, fournir plus de services ou des services plus spécialisés peut entraîner le système dans une situation de conflit.

Principe de la substitution

La substituabilité traitée dans le cadre de notre travail est contextuelle. D'abord, le contexte Ctx doit être calculé sans le composant C_{old} ($Ctx.\mathcal{C} \setminus C_{old}$), c'est-à-dire, calculer l'effet de la suppression du composant C_{old} avec son quadruplet $(C_{old}, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ du contexte Ctx. Ensuite, l'installabilité du composant C_{new} dans le nouveau contexte ($Ctx.\mathcal{C} \setminus C_{old}$) doit être vérifiée par les règles d'installabilité, c'est-à-dire, tous les besoins de C_{new} sont vérifiés et tous ses services fournis ne sont pas en conflits avec Ctx' (cf. la règle CTRIV de la figure 8). Une fois l'installation du nouveau composant est possible dans le nouveau contexte, l'effet de son installation est calculé à partir de sa description de dépendances en utilisant les règles d'installation présentées dans la figure 11 de l'annexe. Par la suite, les services fournis par C_{old} sont comparés avec ceux fournis par C_{new}. C_{old} est substituable dans deux cas : soit l'ensemble des services fournis auparavant par C_{old} et qui ne sont plus fournis par C_{new} est vide, soit tous les services de cet ensemble ne sont pas utilisés obligatoirement par d'autres composants dans le contexte.

La description formelle de la substituabilité contextuelle ainsi que des exemples illustratifs sont présentés en détail dans [5].

Souvent dans les travaux existants [15, 11, 9], la condition de substitution est restrictive car elle consiste à exiger du nouveau composant le même comportement que l'ancien. Dans notre approche le niveau d'abstraction est supérieur car nous nous sommes intéressés à la vérification des propriétés globales du système sans se préoccuper de la vérification de la compatibilité au niveau sémantique et comportemental. Notre vérification pourra être en amont des autres vérifications pour éliminer les cas qui ne sont pas substituables au niveau interface. Les règles de vérification de la substituabilité se basent sur la vérification des nouveaux besoins, le calcul de l'effet de l'ajout du nouveau composant dans le contexte en préservant la consistance et la cohérence du système. Par exemple, dans le cas où le système veut maintenir le niveau de sécurité, l'installation d'un service qui peut influencer le niveau de sécurité du système (le diminuer) est interdite (comme le service `ftp` par exemple). La flexibilité se caractérise par la possibilité de remplacer un composant par un autre selon les exigences du système et les besoins des utilisateurs. Par exemple, dans le cas de ressources limitées un composant peut être remplacé par un autre composant qui consomme moins de ressources.

5. PREUVE DE LA RÉUSSITE ET DE LA SÛRETÉ DU DÉPLOIEMENT

Après avoir proposé un langage de description de dépendance formel et un moteur de raisonnement pour la vérification du déploiement, nous devons *prouver* les propriétés attendues du système de déploiement. Les propriétés qui nous intéressent dans notre système de déploiement sont la *réussite* et la *sûreté*. Cette section décrit briévement ces propriétés pour chacune des opérations d'installation et de désinstallation. Ces preuves sont présentées en détail dans [2].

Tout au long de ce cycle de vie, l'intégrité des sites cibles doit être conservée. Cette cohérence est définie par deux propriétés : la propriété de *réussite* et de *sûreté*. La propriété de *réussite* permet d'assurer que l'application est déployée correctement est donc fonctionnera sur le site client,

comme cela a été prévu. La propriété de *sûreté* permet à une application déployée de ne pas détruire ou perturber, par effet de bord, les applications déjà installées. Par exemple, le partage des librairies DLL entre plusieurs applications peut provoquer de tels effets.

Les propriétés de *réussite* et de *sûreté* doivent être décrites formellement et prouvées pour chaque opération de déploiement. En effet, chaque opération possède un effet sur le système cible donc une vérification et une preuve appropriée. Nous nous sommes inspiré de [18] pour définir nos propres propriétés sans utiliser les politiques d'installation. Cependant, nos propriétés sont définies et prouvées au niveau de l'installation et la désinstallation également. De plus, l'approche utilisée pour la preuve de ces propriétés est différente car nous nous sommes basés sur notre système de règles de déploiement défini dans la section 4.2 pour prouver ces propriétés.

5.1 La propriété de réussite

La propriété de réussite concerne la vérification de certaines conditions sur le composant en question et pas sur le système cible. Dans le cas de l'installation, cette propriété permet d'assurer qu'une fois les contraintes de déploiement sont vérifiées, le déploiement mènera bien à un état dans lequel l'entité déployée est utilisable (ses exigences sont toujours satisfaites et ses services obligatoires sont toujours fournis) dans le nouveau contexte Ctx_{new}.

5.2 La propriété de sûreté

Cette propriété a pour but de fournir l'assurance que le site sera dans un état cohérent après chaque opération de déploiement. Elle concerne la vérification de certaines conditions sur tous les composants existants après le déploiement d'un composant. Pour cela, il faut vérifier les composants et leurs dépendances. Pour chaque composant C du système après une opération de déploiement portant sur un composant A, il faut vérifier son installabilité dans le contexte privé de C ($Ctx_{new} \setminus C$) et vérifier son installation (ses services fournis obligatoirement sont toujours fournis).

6. EXTENSION DU SYSTÈME PAR LES PROPRIÉTÉS

Dans cette section nous présentons une extension du moteur de raisonnement par l'intégration des propriétés. Ces dernières sont décrites dans notre méta-modèle présenté dans la section 3. Elles décrivent l'ensemble des caractéristiques des composants, des services et du système cible. Elles peuvent être fonctionnelles ou non-fonctionnelles. Les descriptions de dépendances, du contexte ainsi que les règles d'installation, de désinstallation et de mise à jour que nous avons proposées doivent être étendues. Ces règles sont décrites en détail dans [2].

6.1 Principe de la description

Les propriétés sont caractérisées par un nom et une valeur. Elles peuvent être associées à un composant ou un service. Elles sont typées, donc leurs valeurs ont un type particulier. Les types supportés sont les nombres, les chaînes de caractères et les booléens. Un composant peut avoir un ensemble de propriétés. Ces propriétés peuvent être relatives à ses services fournis ou le concernant directement. Ainsi la syntaxe de la définition des services et des composants fournis est étendue pour leur ajouter des ensembles de propriétés. Par exemple, un service s avec une propriété p de valeur v est noté $s[p = v]$. Un ensemble de propriétés sera manipulé comme un environnement, il est donc noté $\mathcal{E}$. Le *domaine* de cet environnement représente alors l'ensemble des noms des propriétés qu'il contient :

$$\begin{cases} dom([]) = \varnothing \\ dom([p_1 = v_1, \ldots, p_n = v_n] = \{p_1, \ldots, p_n\} \end{cases}$$

La syntaxe des exigences est également étendue pour permettre la spécification des contraintes sur les propriétés des composants ou services *requis*. Une *contrainte de propriété* φ est définie de manière similaire aux propriétés mais en utilisant en plus de l'égalité d'autres opérateurs de comparaison ($>$, $\geq$, $<$, $\leq$, $=$, $\neq$). Le domaine d'une contrainte de propriété représente le domaine de cette propriété, il est défini comme suit :

$$\begin{cases} dom([]) = \varnothing \\ dom([p_1 O_1 v_1, \ldots, p_n O_n v_n] = \{p_1, \ldots, p_n\} \end{cases}$$

Définition 6 ***(Satisfiabilité)*** *Une contrainte φ peut être satisfaite par un environnement $\mathcal{E}$. Pour cela, il faut (1) que toutes les propriétés de φ aient une valeur dans $\mathcal{E}$ et (2) la valeur d'une propriété dans $\mathcal{E}$ satisfasse toutes les contraintes la concernant dans φ. La* satisfiabilité *est notée par : $\varphi \leftarrow \mathcal{E}$ et définie comme suit : $\varphi \leftarrow \mathcal{E} = (dom(\varphi) \setminus dom(\mathcal{E}) = \varnothing) \wedge (\mathcal{E}(p_1) O_1 v_1) \wedge \cdots \wedge (\mathcal{E}(p_n) O_n v_n)$*
avec $\varphi = [p_1 O_1 v_1, \ldots, p_n O_n v_n]$
$[] \leftarrow \mathcal{E} = true$

La notion de satisfiabilité est utilisé lors de la comparaison d'une exigence d'un service requis avec une propriété d'un service fourni (cf. définition 6). Par exemple, si la contrainte φ vaut $[version \geq 3]$ et $\mathcal{E}$ est égal à $[version = 4]$. La fonction de satisfiabilité $\leftarrow$ consiste à remplacer $version$ dans φ par sa valeur dans $\mathcal{E}$. $\varphi \leftarrow \mathcal{E}$ vaut alors $4 \geq 3$ qui est vraie.

6.2 Description des dépendances

Les dépendances sont décrites selon une grammaire qui étend celle présentée dans la page en ajoutant des propriétés au niveau des services fournis et des contraintes sur les propriétés au niveau des composants et services requis ou interdits.

Le fait d'ajouter un ensemble de propriétés implique la possibilité d'avoir plusieurs instances de composants ou de services. En effet, nous pouvons avoir les mêmes composants ou les mêmes services mais avec des propriétés différentes donc plusieurs instances possibles. Par exemple, la figure 9

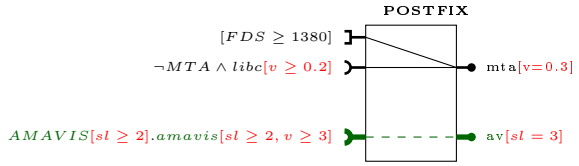

Figure 9: Description des intra-dépendances de `POSTFIX`

présente une extension de la description des intra-dépendances du composant `POSTFIX` par les propriétés. Nous retrouvons des contraintes sur des propriétés au niveau des exigences du composant et des propriétés au niveau des services

fournis. Dans cet exemple, la librairie requise `libc` doit avoir une version supérieure ou égale à 0.2 pour avoir le service `mta` avec une version (`v=0.3`). Le service `av` avec le niveau de sécurité (`sl=3`) peut être fourni par le composant `POSTFIX` si le service `amavis` avec le niveau de sécurité (`sl>=2`) et la version (`v>=3`) est fourni par le composant `AMAVIS` avec un niveau de sécurité (`sl>=2`). `POSTFIX` peut également fournir deux instances du service `mta`.

Pour identifier un composant, nous utilisons le couple (c, num) où c est son nom et num son numéro. Ce numéro est initialisé à 1 et incrémenté à chaque nouvelle instance du composant c (du même nom) installée (cf. la définition 7). La fonction $CalcNum$ utilise une nouvelle forme du contexte différente de celle présentée dans la section 4.1.3.

Définition 7 ***(CalcNum)*** *La fonction CalcNum calcule l'identité d'un composant. elle est définie comme suit :*
$CalcNum(Ctx, c) =$
$max(\{0\} \cup \{n | (c, n, -, -, -, -) \in Ctx.\mathcal{C}\}) + 1$

Un composant peut fournir plusieurs fois le même service avec des propriétés différentes. Il faut donc identifier ces services pour les différencier. L'identité d'un service est notée id_s, c'est un quadruplet $(c, num, s, \mathcal{E}_s)$, où le couple (c, num) représente l'identité de son fournisseur et $\mathcal{E}_s$ ses propriétés.

6.3 Description du contexte

Le *Contexte* a une forme similaire à celle du système initial, il est juste enrichi pour mémoriser toutes les informations concernant les propriétés et les contraintes. Chaque composant est constitué de l'ensemble des éléments (c,num, $\mathcal{E}_c$,$\mathcal{P}_s$,$\mathcal{F}_s$,$\mathcal{F}_c$) décrivant : c son nom, num son numéro, $\mathcal{E}_c$ l'ensemble de ses propriétés, $\mathcal{P}_s$ l'ensemble de ses services fournis avec leurs propriétés $(s, \mathcal{E}_s)$, $\mathcal{F}_s$ l'ensemble de ses services interdits avec leurs contraintes (s, φ_s) et $\mathcal{F}_c$ l'ensemble de ses composants interdits avec leurs contraintes (c, φ_c)

Un nœud du graphe $\mathcal{G}$ représente l'identité d'un service disponible ($id_s = (id_c, s, \mathcal{E}_s)$), où $id_c = (c, num)$ représente l'identité de son composant (fournisseur) et $\mathcal{E}_s$ représente ses propriétés. Un arc est le couple de nœuds $id_{s1} \overset{x,\varphi_c,\varphi_s}{\longmapsto} id_{s2}$, où id_{s2} requiert id_{s1} avec la contrainte φ_c sur le composant qui fournit le service s_2 et la contrainte φ_s sur le service requis id_{s1}. Chaque arc est étiqueté également avec le type de la dépendance (au dessus de la flèche, le x), soit obligatoire M (mandatory) soit optionnel O(optional). Les contraintes sont conservées dans le graphe pour les prendre en compte dans le cas de la vérification d'autres opérations de déploiement. Par exemple, si un composant a la propriété $(NS = 3)$ est utilisé par un autre avec une contrainte sur son niveau de sécurité $(NS \geq 3)$, ce dernier ne pourra pas être mis à jour par un autre composant qui a la propriété $(NS = 2)$. Pour cela, toutes les contraintes relatives aux composants et aux services sont mémorisées dans le graphe de dépendance.

6.4 Description de l'installation

La différence avec le premier système est la prise en compte des propriétés et des contraintes exigées sur les composants et les services. Nous manipulerons donc les identifiants de composants et services. La première étape est la vérification de l'*installabilité* du composant. Elle consiste à (1) vérifier que toutes les contraintes imposées sur les propriétés des services requis par ce composant sont disponibles dans le contexte (2) s'assurer qu'aucun des composants à installer avec leurs propriétés n'est interdit et tous les services qu'il fournit avec leurs propriétés ne font pas partie de l'ensemble des services interdits.

La deuxième étape consiste à calculer l'*effet* de l'installation du composant sur le système.

6.5 Description de la désinstallation

Le principe de vérification est exactement le même que celui du premier système. La différence réside dans la syntaxe des dépendances et la gestion des identifiants (*instances*) des composants et des services. En effet, nous pouvons désinstaller une instance d'un composant (c, num) si aucune de ses instances de services fournis id_s n'est utilisée obligatoirement par d'autres composants. Pour cela, pour chaque instance d'un service fourni, il faut s'assurer qu'elle ne possède aucune dépendance obligatoire avec d'autres instances de services. Ainsi, un service peut être désinstallé s'il n'est pas utilisé obligatoirement mais s'il est utilisé directement ou indirectement de façon optionnelle, il faut s'assurer que les services qui en dépendent de façon optionnelle directement ou indirectement ne possèdent pas d'exigences dans le graphe. La principale différence est dans la forme du graphe de dépendances. En effet, les nœuds représentent les identifiants des services et les arcs sont étiquetés par le type de dépendance, les contraintes du composant et du service.

L'effet d'une désinstallation est calculé de la même façon que dans le premier système. La différence réside dans la gestion des identifiants des composants et des services. Nous pouvons désinstaller une instance d'un composant avec toutes ses instances de services sans désinstaller les autres instances.

6.6 Description de la mise à jour

Pour décider si un composant C_{old} peut être remplacé par un composant C_{new}, il faut s'assurer que les services qui sont fournis par C_{new} avec leurs propriétés vérifient toujours les contraintes exigées par les autres composants qui les utilisent. La principale différence est donc dans la vérification de la satisfiabilité des contraintes exigées par les propriétés fournies. La comparaison ne se fait pas uniquement par rapport aux services fournis mais également par rapport aux services avec toutes leurs propriétés. Par ailleurs, il faut maintenir la validité des contraintes des autres composants qui dépendaient de ces services.

7. CONCLUSION

Pour palier au manque d'outils de déploiement de composant sûrs et adaptables, il nous a paru fondamental de passer d'abord par une étape de *modélisation* ensuite par une étape de *formalisation* mathématique et enfin par une étape de *preuve* de la réussite et de la sûreté du déploiement (cette étape n'est pas l'objet de ce papier).

Dans l'étape de modélisation nous avons suivi une approche conceptuelle pour proposer un modèle de déploiement générique pour assurer la flexibilité. Ainsi, la spécialisation du modèle est possible et conduit à une adaptation du système selon les besoins. La deuxième étape, concerne la formalisation qui représente la description formelle des entités de déploiement (composant, application et système cible) ainsi que la vérification des opérations de déploiement des composants (installation, désinstallation et mise à jour). Pour la description des composants nous avons proposé un langage d'*intra-dépendance* qui permet d'exprimer la rela-

tion entre chaque service fourni et ses exigences. L'originalité de ce langage est qu'il est expressif et permet d'exprimer des fonctionnalités optionnelles, des choix, etc. De plus, son utilisation permet simplifier la gestion des dépôts de composants. Un prototype associé a cette formalisation a été développé en OCaml. Nous avons également proposé une extension des descriptions formelles des entités de déploiement ainsi que les règles utilisées par la description des propriétés non-fonctionnelles. L'intérêt est de pouvoir personnaliser le déploiement en fonction des exigences du système et des besoins de l'utilisateur pour le rendre plus flexible.

Cependant, il reste de nombreux travaux à explorer. Nous pouvons citer :

- La gestion de l'aspect *dynamique* du déploiement. Dans ce que nous avons proposé le déploiement est statique et le contexte n'est pas réévalué après chaque opération de déploiement. La remise en cause de la décision du déploiement n'est donc pas prise en compte.
- Intégration de la formalisation des *politiques* de déploiement dans le moteur de raisonnement pour personnaliser le déploiement.
- la gestion du déploiement *distribué* et la prise en compte des contraintes de localisation.

8. REFERENCES

[1] D. Ayed. *Déploiement sensible au contexte d'applications à base de composants*. Thèse de doctorat, Université d'Evry-Val d'Essonne, Novembre 2005.

[2] M. Belguidoum. *Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels*. Thèse de doctorat, Ecole Normale Supérieure des Télécommunications de Bretagne, Février 2008.

[3] M. Belguidoum and F. Dagnat. Dependability in software component deployment. In *DepCoS-RELCOMEX*, pages 223–230, Szklarska Poreba, Poland, June 2007. IEEE Computer Society.

[4] M. Belguidoum and F. Dagnat. Dependency management in software component deployment. *Electron. Notes Theor. Comput. Sci.*, 182:17–32, 2007.

[5] M. Belguidoum and F. Dagnat. Formalization of component substitutability. *Electr. Notes Theor. Comput. Sci.*, 215:75–92, 2008.

[6] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Lecture Notes in Computer Science*, 32(7):38–45, July 1999.

[7] P. Brada. *Specification-Based Component Substitutability and Revision Identification*. PhD thesis, Charles University, Prague, August 2003.

[8] A. Carzaniga, A. Fuggetta, R. Hall, A. Hoek, D. Heimbigner, and A. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, April 1998.

[9] I. Cerna, P. Varekova, and B. Zimmerova. Component substituability via equivalencies of component-interaction automata. In *FACS'06-International Workshop on Formal Aspects of Component Software*, Prague, Czech Republic, September 2006. ENTCS.

[10] H. Cervantes. *Vers un modèle de composants orienté services pour supporter la disponibilité dynamique*. Thèse de doctorat, Université Joseph Fourier Grenoble I, Mars 2004.

[11] S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *In 3rd Workshop on Specification and Verification of Component-based Systems*, 2004.

[12] F. Déchelle and F. Mancinelli. EDOS-Tools Tutorial: EDOS Tools for Linux Distributions Dependencies Management and Quality Assurance. In *OSS*, pages 363–364, 2007.

[13] EDOS : Environment for the development and Distribution of Open Source software. `http://www.edos-project.org/xwiki/bin/view/Main/WebHome`.

[14] V. Lestideau. *Modèles et environnement pour configurer et déployer des systémes logiciels*. Thèse de doctorat, Université de Savoie, Décembre 2003.

[15] B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints. MU CS-99-156, School of Computer Science, Carnegie Mellon University, july 1999.

[16] N. Merle. *Architecture pour les systèmes de déploiement logiciel à grande échelle : prise en compte des concepts d'entreprise et de stratégie*. Thèse de doctorat, Université Joseph Fourier, Décembre 2005.

[17] OMG. Deployment and Configuration of Component-based Distributed Applications. Specification version 4, OMG, April 2006.

[18] A. S. Parrish, B. Dixon, and D. Cordes. A conceptual foundation for component-based software deployment. *Journal of Systems and Software*, 57(3):193–200, 2001.

[19] R. Reussner, 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)*, June 2003.

[20] SAT4J : Bringing the power of SAT technology to the Java platform. `http://sat4j.org/`.

[21] G.N. Silva. Apt-howto, 2004. `http://www.debian.org/doc/manuals/apt-howto/`.

[22] J.M. Soucé and L. Duchien. État de l'art sur les langages de description d'architecture. Livrable I, Projet RNTL ACCORD, 2002.

[23] J.A. Stafford and A. L. Wolf. Architecture-level dependence analysis for software systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–451, 2001.

[24] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

9. ANNEXE

9.1 Les composants et services interdits

Les fonctions qui calculent l'ensemble des services disponibles AS, celui des composants disponibles AC, celui des services interdits FS et enfin celui des composants interdits FCsont définies comme suit :

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

avec $Ctx.\mathcal{C}$ représente le quadruplet $(c, \mathcal{P}_s, \mathcal{F}_s, \mathcal{F}_c)$ qui décrit le composant c avec l'ensemble de ses services fournis $\mathcal{P}_s$, l'ensemble de ses services interdits $\mathcal{F}_s$ et l'ensemble de ses composants interdits $\mathcal{F}_c$.

9.2 Le graphe de dépendances

Les règles de génération du graphe de dépendance sont représentées dans la figure 10.

$$\text{GTRUE:}\ Ctx, c, s \vdash_G true \Rightarrow \varnothing$$

$$\text{GAND:}\ \frac{Ctx, c, s \vdash_G P_1 \Rightarrow \mathcal{G}_1 \qquad Ctx, c, s \vdash_G P_2 \Rightarrow \mathcal{G}_2}{Ctx, c, s \vdash_G P_1 \wedge P_2 \Rightarrow \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\text{GORL:}\ \frac{Ctx, c, s \vdash_G Q_1 \Rightarrow \mathcal{G}_1 \qquad Ctx \vdash_P Q_1}{Ctx, c, s \vdash_G Q_1 \vee Q_2 \Rightarrow \mathcal{G}_1}$$

$$\text{GORR:}\ \frac{Ctx \nvdash_P Q_1}{Ctx, c, s \vdash_G Q_1 \vee Q_2 \Rightarrow \mathcal{G}_2}$$

$$\text{GVAR:}\ Ctx, c, s \vdash_G [v\ O\ V] \Rightarrow \varnothing \qquad \text{GNOTS:}\ Ctx, c, s \vdash_G \neg s' \Rightarrow \varnothing$$

$$\text{GNOTC:}\ Ctx, c, s \vdash_G \neg c' \Rightarrow \varnothing$$

$$\text{GSERVC:}\ \frac{(c', \mathcal{P}_s, -, -) \in Ctx.\mathcal{C} \qquad s' \in \mathcal{P}_s}{Ctx, c, s \vdash_G c'.s' \Rightarrow \{c'.s' \overset{\mathsf{M}}{\longmapsto} c.s\}}$$

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

Figure 10: Les règles de calcul du graphe de dépendance

9.2.1 Les regles de vérification de l'installation

Les règles d'installation sont représentées dans la figure 11.

$$\text{ITRIV:}\ \frac{Ctx \vdash P \quad Ctx, c, s \vdash_G P \Rightarrow \mathcal{G} \quad s \notin FS(Ctx) \quad CalcF(P) = \mathcal{F}_s, \mathcal{F}_c}{Ctx, c \vdash_I (P \Rightarrow s) \Rightarrow \{s\}, \mathcal{F}_s, \mathcal{F}_c, \mathcal{G}}$$

$$\text{INOT}_1\text{:}\ \frac{Ctx \nvdash_P P}{Ctx, c \vdash_I (P \Rightarrow s) \Rightarrow \bot} \qquad \text{INOT}_2\text{:}\ \frac{s \in FS(Ctx)}{Ctx, c \vdash_I (P \Rightarrow s) \Rightarrow \bot}$$

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

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

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

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

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

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

$$\text{IOR3:}\ \frac{Ctx, c \vdash_I D_1 \Rightarrow \bot \qquad Ctx, c \vdash_I D_2 \Rightarrow \bot}{Ctx, c \vdash_I D_1 \,\#\, D_2 \Rightarrow \bot}$$

Figure 11: Les règles d'installation

9.3 Exemple de vérification de l'installation

La preuve de l'installabilité

Soit le composant `POSTFIX` avec la dépendance $(D_1 \bullet D_2)$ tel que :

$$\begin{cases} D_1 = [FDS \geq 1380] \wedge ?C_{SM} \wedge S_{lib} \Rightarrow S_{MTA} \\ D_2 = ?(S_{amavis} \Rightarrow S_{AV}) \end{cases}$$

Les services qui sont obligatoirement requis par `POSTFIX` sont : une librairie (S_{lib}) avec la condition de l'absence du composant `SENDMAIL` (C_{SM}), le service `amavis` (S_{amavis}) qui fournit l'antivirus S_{AV} est optionnel.

Il faut donc prouver la conjonction des dépendances D_1 et D_2 ($Ctx \vdash D_1 \bullet D_2$). La dépendance D_2 représente une dépendance optionnelle donc elle est toujours vérifiée pendant cette étape. La preuve de l'installabilité `POSTFIX` est donc liée à la dépendance D_1. La démonstration est illustrée ci-dessous en respectant les règles de véfication de l'installabilité la figure 8.

$$\text{A} = \frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{A}}{Ctx \vdash_C [FDS \geq 1380] \wedge ?C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}} \quad Ctx \vdash_C ?(S_{amavis} \Rightarrow S_{AV})}{Ctx \vdash_C D_1 \bullet D_2} \qquad C_{PX} \notin \varnothing}{Ctx \vdash C_{PX} : D_1 \bullet D_2}}{}}{}}$$

$$\text{A} = \frac{\dfrac{\dfrac{500000 \geq 1380}{Ctx \vdash_P FDS \geq 1380} \quad \dfrac{C_{SM} \notin \{C_1, C_A\}}{Ctx \vdash_P ?C_{SM}} \quad \dfrac{S_{lib} \in \{S_{lib}, S_{amavis}\}}{Ctx \vdash_P S_{lib}}}{Ctx \vdash_P [FDS \geq 1380] \wedge ?C_{SM} \wedge S_{lib}} S_{MTA} \notin \varnothing}{Ctx \vdash_C [FDS \geq 1380] \wedge ?C_{SM} \wedge S_{lib} \Rightarrow S_{MTA}}$$

Dans notre contexte, le composant `POSTFIX` est donc installable.

La preuve de l'installation

L'étape d'installation suit l'étape de vérification de l'installabilité, elle consiste à calculer l'effet de l'ajout du composant `POSTFIX` (C_{PX}) dans le contexte. Cet effet consiste à mettre à jour l'ensemble des services disponibles, l'ensemble des services interdits, l'ensemble des composants interdits et le graphe de dépendance.

La dérivation de l'installation de D_1 est notée **A'**, le prédicat de cette dépendance est représenté par P. Le résultat de l'installation de D_1 est caractérisé par l'ajout du composant C_{PX} avec le service S_{MTA} dans l'ensemble de ses services fournis, l'ajout du composant C_{SM} dans l'ensemble de ses composants interdits et la construction du graphe de dependance $\mathcal{G}_1$. La dépendance optionnelle D_2 va être vérifiée pendans cette étape pour fournir les services qui peuvent

êtres disponibles. La dérivation de D_2 est représentée par l'arbre de dérivation **B**. L'effet de la dépendance optionnelle consiste à ajouter le service S_{AV} dans l'ensemble $\mathcal{P}_s^2$ du composant C_{PX} et construire le graphe $\mathcal{G}_2$ résultant. La preuve de l'installation est réalisée par les règles de dérivation de la figure 11 :

$$\textsc{IComp}\ \frac{\textsc{IAnd3}\ \dfrac{\textbf{A'}\quad \dfrac{}{Ctx, C_{PX} \vdash_I D_1 \Rightarrow \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1, \mathcal{G}_1} \qquad \textbf{B}\quad \dfrac{}{Ctx, C_{PX} \vdash_I D_2 \Rightarrow \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2, \mathcal{G}_2}}{Ctx, C_{PX} \vdash_I D_1 \bullet D_2 \Rightarrow \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2}}{Ctx \vdash_C C_{PX} : D_1 \bullet D_2 \Rightarrow \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2}$$

$$\begin{cases} \mathcal{P}_s^1, \mathcal{F}_s^1, \mathcal{F}_c^1, \mathcal{G}_1 = \{S_{MTA}\}, \varnothing, \{C_{SM}\}, \{C_1.lib \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\} \\ \mathcal{P}_s^2, \mathcal{F}_s^2, \mathcal{F}_c^2, \mathcal{G}_2 = \{S_{AV}\}, \varnothing, \varnothing, \{A.S_{amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}\} \\ \mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2, \mathcal{G}_1 \cup \mathcal{G}_2 = \{S_{MTA}, S_{AV}\}, \varnothing, \{C_{SM}\}, \{C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}, A.S_{amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}\} \end{cases}$$

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

$$\textbf{B=IOpt2}\ \frac{\dfrac{\dfrac{S_{amavis} \in \{S_{lib}, S_{amavis}\}}{Ctx, C_{PX}, S_{AV} \vdash_G S_{amavis} \Rightarrow \{C_A.S_{amavis} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{AV}\}} S_{AV} \notin \varnothing \qquad CalcF(S_{amavis}) = \varnothing, \varnothing}{Ctx, C_{PX} \vdash_I (S_{amavis} \Rightarrow S_{AV}) \Rightarrow \{S_{AV}\}, \varnothing, \varnothing, \{C_A.S_{amavis} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{AV}\}}}{Ctx, C_{PX} \vdash_I ?(S_{amavis} \Rightarrow S_{AV}) \Rightarrow \{S_{AV}\}, \varnothing, \varnothing, \{C_A.S_{amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}\}}$$

Après l'installation du composant C_{PX}, le contexte est mis à jour et devient :

$$\begin{cases} \boldsymbol{\mathcal{E}} = \{FDS = 500000, OS = LINUX, RAM = 128\}; \\ \boldsymbol{\mathcal{C}} = \{(C_1, S_{lib}, \varnothing, \varnothing), (C_A, \{S_{amavis}\}, \varnothing, \varnothing), (C_{PX}, \{S_{MTA}, S_{AV}\}, \varnothing, \{C_{SM}\})\}, \\ \boldsymbol{\mathcal{G}} = \{A.S_{amavis} \overset{\mathsf{O}}{\longmapsto} C_{PX}.S_{AV}, C_1.S_{lib} \overset{\mathsf{M}}{\longmapsto} C_{PX}.S_{MTA}\} \end{cases}$$