

Formalisation de l'expression d'un plan de déploiement autonomique à base de contraintes

Raja Boujbel, Sébastien Leriche, Jean-Paul Arcangeli, Oudom Kem

▶ **To cite this version:**

Raja Boujbel, Sébastien Leriche, Jean-Paul Arcangeli, Oudom Kem. Formalisation de l'expression d'un plan de déploiement autonomique à base de contraintes. Journées francophones Mobilité et Ubiquité - UBIMOB 2014, Jun 2014, Nice, France. pp. 1-8. hal-01135234

HAL Id: hal-01135234
https://hal.science/hal-01135234

Submitted on 25 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Open Archive TOULOUSE Archive Ouverte (OATAO)
OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in : http://oatao.univ-toulouse.fr/
Eprints ID : 13091

To cite this version : Boujbel, Raja and Leriche, Sébastien and Arcangeli, Jean-Paul and Kem, Oudom *Formalisation de l'expression d'un plan de déploiement autonomique à base de contraintes*. (2014) In: Journées francophones Mobilité et Ubiquité - UBIMOB 2014, 5 June 2014 - 6 June 2014 (Nice, France).

Any correspondance concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr

Formalisation de l'expression d'un plan de déploiement autonomique à base de contraintes

Raja Boujbel[1], Sébastien Leriche[2], Jean-Paul Arcangeli[1], Oudom Kem[1]
[1]Université de Toulouse - IRIT UPS 118 route de Narbonne F-31062 Toulouse, France
<prénom>.<nom>@irit.fr
[2]Université de Toulouse - ENAC 7 av. Edouard Belin 31055 Toulouse, France
<prénom>.<nom>@enac.fr

RÉSUMÉ

Les systèmes ambiants sont devenus massivement distribués. Le nombre d'appareils hétérogènes, et la variété de composants logiciels à déployer sur ces systèmes pour en assurer le bon fonctionnement ne cessent de croître. Leur topologie est en évolution constante, liée à l'apparition et la disparition des dispositifs mobiles. De ce fait, le déploiement de logiciel dans ces systèmes est un problème ouvert. Notre approche pour diminuer la complexité de cette opération, est le déploiement autonomique. Dans cet article, nous partons d'un langage dédié (DSL) nommé MuScADeL, pour lequel nous proposons une formalisation de l'expression du déploiement autonomique. Ensuite, nous montrons comment traduire les propriétés de déploiement en un problème de satisfaction de contraintes, et comment nous obtenons un plan de déploiement conforme qui sera enfin interprété par un intergiciel de déploiement autonomique.

Mots-Clés

Systèmes ambiants, Mobilité, Déploiement autonomique

1. INTRODUCTION

Le déploiement d'un système ambiant à grande échelle est un processus complexe qui implique la gestion d'un très grand nombre de machines hétérogènes, souvent mobiles, impliquant une topologie variable et dynamique. Ce processus a pour objectif de rendre un logiciel disponible pour l'utilisation, puis de le maintenir dans un état opérationnel. Il comprend un certain nombre d'activités liées [2, 4], telles que l'installation du logiciel dans son environnement d'exécution (transfert et configuration), l'activation, la mise à jour, la reconfiguration, la désactivation et la désinstallation.

Nous appelons *domaine de déploiement* l'ensemble des appareils distribués sur les réseaux et pouvant héberger les composants du système logiciel. Un *plan de déploiement* est une correspondance entre le système de composants et le domaine de déploiement, complétée par les données de configuration des composants logiciels. À l'exécution, le logiciel est déployé sur les appareils hôtes conformément au plan de déploiement.

Le déploiement de systèmes ambiants à grande échelle est souvent trop complexe à réaliser pour un opérateur humain. Il est donc nécessaire de repenser ce processus. Il doit répondre à des exigences et à des contraintes émanant de différentes parties prenantes et portant à la fois sur le logiciel à déployer et les machines cibles, en particulier sur leur distribution et leur dynamique. Nous souhaitons aussi pouvoir prendre en compte des exigences caractérisées par des expressions multi-échelles [16]. Concernant la mobilité et l'ouverture, le déploiement doit réagir à l'instabilité du réseau de machines, c'est-à-dire aux connexions et déconnexions et aux variations de la disponibilité et de la qualité des ressources. Ainsi, le déploiement à grande échelle doit être un processus continu qui supporte l'exécution du logiciel et nécessite une certaine adaptabilité.

L'autonomie vise à éviter (ou à limiter) les interventions humaines dans la gestion de l'exécution du déploiement. L'approche de l'informatique autonome [10] où le système auto-gère certaines propriétés (auto-configuration, auto-réparation...) nous semble apporter une réponse à certaines exigences du déploiement des systèmes logiciels distribués. C'est cette approche que nous appelons « *déploiement autonomique* » [14] et que nous poursuivons dans le cadre du projet ANR ICOME [1]. Nous envisageons l'automatisation et l'autonomie du déploiement à plusieurs niveaux, dont seul le premier est traité dans cet article :

- d'une part, à travers la génération du plan de déploiement à partir de spécifications de haut niveau exprimées par le concepteur du déploiement ;
- d'autre part, en confiant le déploiement, y compris le maintien du système déployé dans un état opérationnel, à un système de déploiement autonome.

Cet article est organisé comme suit. Dans la section 2, nous présentons un état de l'art des systèmes de déploiement. Dans la section 3, nous rappelons le processus de génération du plan de déploiement, puis dans la section 4, quelques éléments de notre langage dédié, MuScADeL. Dans la section 5, nous présentons la formalisation des propriétés de déploiement. Enfin, dans la section 6, nous montrons comment exploiter cette formalisation au travers d'un exemple complet.

Copyright is held by the author/owner(s)
UbiMob 2014, 5–6 juin 2014, Nice, France.
.

2. ÉTAT DE L'ART

La complexité du déploiement apparaît dans plusieurs travaux, qui visent à diminuer l'intervention humaine dans tous le processus. Dans Software Dock [8] et QUIET [13], les auteurs proposent un déploiement automatique distribué dont les architectures sont basés sur les agents mobiles, comme support de l'autonomie dans la réalisation du déploiement. Disnix [21] propose une autre solution pour l'automatisation du déploiement basée sur des modèles.

Une autre problématique est celle des appareils mobiles à capacités limitées. Ils sont sujets aux déconnexions, offrent une qualité de service variable, et par conséquent demandent un déploiement plus spécifique. Kalimucho [12] propose une adaptation du plan de déploiement en fonction de la qualité de service. Codewan [7] supporte un déploiement opportuniste d'application à base de composants sur les réseaux déconnectés MANET. Cloudlet [17] est une solution à base de machine virtuelle qui permet le transférer la charge de calcul d'un appareil mobile à un cloud de proximité.

La conception du déploiement est une tâche difficile. Les concepteurs doivent prendre en compte les différentes activités, ainsi que les propriétés des applications et du domaine de déploiement. De plus, exprimer directement un plan de déploiement n'est pas toujours la bonne solution, et parfois une tâche impossible humainement. Certains travaux permettent une couche d'abstraction et facilitent l'expression du déploiement. DeployWare [6] est un *framework* pour le déploiement à large échelle basé sur un langage de modélisation dédié au déploiement. Un autre *framework*, ADME [5], cible le déploiement autonomique et se base sur une résolution d'un problème de satisfaction de contrainte. TUNe [20] fournit un formalisme de haut niveau pour la gestion autonome de grilles à grande échelles décentralisées. Enfin, j-ASD [14] est un *framework* de déploiement autonomique contenant un langage dédié au déploiement sur des systèmes P2P ou des grilles, à partir duquel nous basons nos travaux.

3. PROCESSUS DE GÉNÉRATION

Cette section rappelle succinctement notre vision du processus de déploiement autonomique.

Dans un premier temps, le concepteur du déploiement spécifie, au moyen du langage dédié MuScADeL, les composants à déployer ainsi qu'un ensemble de propriétés de déploiement que le système doit posséder. Ces propriétés représentent à la fois les contraintes propres aux composants (conditions logicielles et matérielles à respecter) et les exigences de déploiement du concepteur. Le plan de déploiement doit satisfaire ces propriétés étant donné l'état courant du domaine de déploiement. Ainsi, le plan de déploiement est une ou la solution d'un problème qui se présente naturellement comme un problème de satisfaction de contraintes. La production du plan de déploiement est donc supportée par un solveur de contraintes (le choix du solveur, un composant sur étagère, est discuté en section 6.3).

Pour pouvoir être traité par le solveur de contraintes, les différentes informations issues de l'analyse du descripteur MuScADeL et de l'état du domaine doivent être exprimées sous la forme de « contraintes ». À partir d'ici, le terme contrainte sera employé en ce sens : il fera donc référence à l'ensemble de ces informations (propriétés de déploiement, état du domaine) et pas seulement aux contraintes propres aux composants.

La génération du plan de déploiement suppose donc la transformation des différentes informations en contraintes préalablement à l'appel au solveur. Ainsi, une activité de formalisation transforme les propriétés en contraintes. Cette formalisation constitue l'ensemble des contraintes qui définissent le problème et sont données en entrée du solveur. La figure 1 représente le processus de génération du plan de déploiement sous la forme d'un diagramme SPEM.

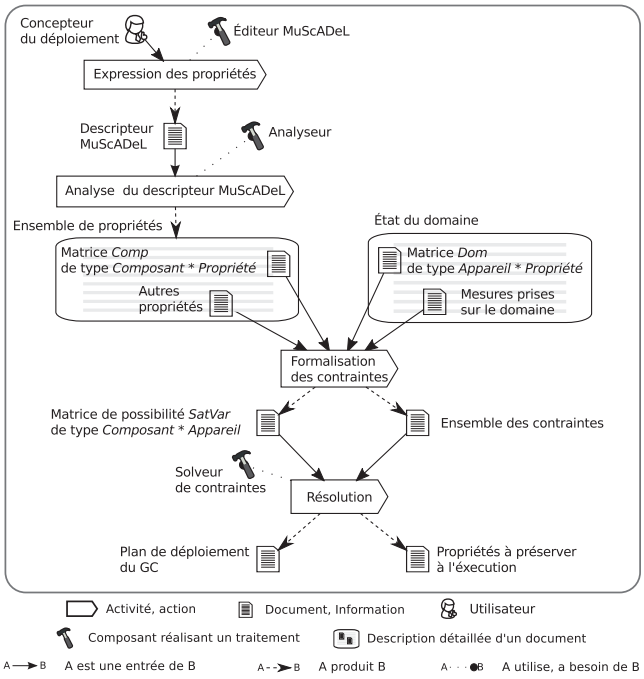


Figure 1: Processus de génération du plan de déploiement

4. MuScADeL

Dans cette section, nous présentons notre DSL dédié au déploiement autonomique de systèmes multi-échelles, appelé MuScADeL (***Mu**lti**Sc**ale **A**utonomic **De**ployment **L**anguage*). Un exemple de code MuScADeL est donné dans la section 6, dans le listing 1.

Avec MuScADeL, le concepteur du déploiement peut exprimer des propriétés de déploiement, tel que des choix de conception, les exigences de déploiement, et des contraintes propres aux composants. Il est possible d'exprimer des propriétés de déploiement d'une application monolithique ou d'une application à base de composants sur un domaine de déploiement, qui peut être composé de centaines d'appareils. MuScADeL permet d'avoir une couche d'abstraction, ce lui permet d'être utilisé sans une grande expertise dans la réalisation des activités de déploiement. La grammaire de MuScADeL est définie selon une syntaxe EBNF [1].

En utilisant MuScADeL, le concepteur du déploiement peut définir plusieurs entités :

- Des composants avec le mot-clé `Component`,
- des sondes avec le mot-clé `Probe`,
- des critères avec le mot-clé `BCriterion`,

1. La grammaire complète est disponible à l'adresse suivante : http://www.anr-income.fr/T5/muscadel-ebnf.html

– des sondes multi-échelles avec le mot-clé `MultiScaleProbe`,
– et le déploiement avec le mot-clé `Deployment`.

Les composants sont nommés et doivent contenir la version, l'adresse à laquelle composant est téléchargeable, et optionnellement, les composants requis et les contraintes du composant. Ces contraintes sont les conditions logicielles et matérielles dont le composant à besoin pour un fonctionnement nominal. Elles doivent êtres satisfaites lors de la génération du plan de déploiement, ainsi que lors de l'exécution.

Les sondes logicielles permettent de récupérer des informations logicielles et matérielles concernant le domaine de déploiement. Les sondes sont installées sur tous les appareils du domaine de déploiement.

Les critères (ici, il s'agit de critères de base par opposition aux critères multi-échelles, introduits plus tard) sont un ensemble de conditions à respecter. Un critère peut être utilisé pour définir une contrainte portant sur un composant ou une exigence de déploiement. Il est possible de définir plusieurs conditions dans un critère, et chaque condition fait appel à une sonde. Une condition peut être soit un test d'une valeur de la sonde utilisée, soit un test de l'existence ou de l'activité de la sonde utilisée.

Les sondes multi-échelles permettent de récupérer des informations multi-échelles concernant l'appareil cible.

La partie déploiement permet de spécifier les exigences de déploiement. Ces exigences peuvent prendre plusieurs formes : nombre d'occurrences, critère sur un composant, critère multi-échelle. Le nombre d'occurrences peut être précisé : soit fixe, déploiement sur deux appareils ; soit un intervalle, déploiement sur au deux à quatre appareils ; soit maximal, déploiement sur tous les appareils pouvant héberger le composant, y compris ceux qui entrent dans le domaine de déploiement à l'exécution de l'application (`All`) ; soit par rapport au nombre d'instances d'un autre composant, déploiement d'une instance d'un composant pour chaque 3 instances d'un autre composant précisé. Les critères sur les composants sont spécifiés en utilisant le critère définit par le mot-clé `BCriterion`. Les critères multi-échelles peuvent prendre plusieurs formes : spécification d'une échelle, spécification d'une instance d'échelle, spécification d'une instance d'échelle par rapport à un autre composant, par exemple, déploiement d'un composant sur la même instance d'échelle qu'un autre composant (`SameValue`), et spécification de déploiement d'une instance d'un composant par instance d'échelle, par exemple, un composant par réseau local (`Each`).

5. FORMALISATION DES CONTRAINTES

Dans cette section, nous commençons par présenter les structures de données utilisées, puis la formalisation des propriétés de déploiement.

5.1 Données et structure de données

5.1.1 Données en entrée

L'analyse du descripteur MuScADeL a permis d'identifier un certain nombre de propriétés que le système doit posséder. Elle a produit une matrice $Comp$ de type $Composant * Propriété$ telle que [2] :

- $Comp(i, j) = 1$ si le déploiement du composant $Composant_i$ est contraint par la propriété $Propriété_j$,
- $Comp(i, j) = 0$ sinon.

Ici, on n'a considéré que des propriétés simples, c'est-à-dire ne concernant qu'un seul composant. D'autres propriétés, qui ne sont pas prises en compte dans la matrice $Comp$, restent à considérer.

D'autre part, indépendamment de l'analyse du descripteur MuScADeL, l'analyse de l'état du domaine a produit une matrice Dom de type $Appareil * Propriété$ telle que :

- $Dom(i, j) = 1$ si l'appareil $Appareil_i$ possède la propriété $Propriété_j$,
- $Dom(i, j) = 0$ sinon.

Il faut noter que les sondes, y compris les sondes multi-échelles, sont utilisées pour construire la matrice Dom. De manière générale, les mesures prises par les sondes sur les appareils font aussi partie de l'état du domaine. Elles sont fournies sous la forme d'une table associant appareil et mesure.

5.1.2 Données en sortie

Le plan de déploiement produit par le solveur se présente sous la forme d'une matrice d'obligation $Oblig$ de type $Composant * Appareil$, qui définit le placement des composants, telle que

- $Oblig(i, j) = 1$ si le composant $Composant_i$ doit être déployé sur l'appareil $Appareil_j$,
- $Oblig(i, j) = 0$ sinon.

5.2 Formalisation des contraintes

5.2.1 Contraintes et exigences

Une matrice de possibilité $SatVar$, de type $Composant * Appareil$, est construite. Chaque coefficient de $SatVar$ est une variable qui peut prendre sa valeur dans $\{0, 1\}$.

À partir des matrices $Comp$ et Dom, des contraintes sont ajoutés sur certains coefficients. Ces contraintes sont l'affectation à 0 du coefficient, correspondant à une impossibilité pour un appareil d'héberger le composant. Cela se traduit par la contrainte suivante [3] (nb_app et nb_comp correspondent respectivement au nombre d'appareils et au nombre de composants impliqués dans le déploiement) :

$$\forall i \in \{1, .., nb_comp\}, \forall j \in \{1, .., nb_app\}$$
$$Comp(i) \cdot Dom(j) = \vec{0} \implies SatVar(i, j) = 0 \quad (1)$$

Ici, $Comp(i)$ et $Dom(j)$ représentent respectivement les lignes i et j des matrices $Comp$ et Dom, l'opérateur $\cdot$ construit une ligne composée des produits deux à deux des éléments de deux lignes données, et $\vec{0}$ représente le vecteur nul.

5.2.2 Nombre d'instances

Cardinalité.

Pour chaque composant (une ligne de la matrice $SatVar$), le nombre d'instances est défini par la valeur de la somme des éléments de la ligne. On construit ainsi une contrainte pour chaque composant en fonction du nombre d'instances.

Ainsi, si le composant C_k doit être déployé sur n_k appa-

2. Les composants dont on a spécifié qu'ils sont requis par des composants à déployer sont pris en compte ici et intégrés à la matrice $Comp$.

3. Par convention, les indices de lignes et de colonnes des matrices et des tableaux commencent à 1.

reils, cela se traduirait par la contrainte :

$$\sum_{j=1}^{nb_app} SatVar(k,j) = n_k \quad (2)$$

Si le composant C_k doit être déployé sur n_k à m_k appareils, cela se traduirait par la contrainte :

$$n_k \leq \sum_{j=1}^{nb_app} SatVar(k,j) \leq m_k \quad (3)$$

All.

La cardinalité All spécifie qu'un composant doit être déployé sur tous les appareils qui peuvent l'héberger. Il faut maximiser le nombre d'appareils qui peuvent héberger le composant. L'expression `Ck @ All` se traduirait par la formule suivante :

$$\max_{\substack{j\in\{1,..,nb_app\}\\ SatVar(k,j)}} \left(\sum_{i=1}^{nb_app} SatVar(k,i) \right) \quad (4)$$

Ratio.

Un ratio entre les nombres d'instances de différents composants peut se traduire en s'appuyant sur les mêmes principes, mais en associant plusieurs lignes de $SatVar$. L'expression `Ck @ n/m Cl` se traduirait par la contrainte :

$$\sum_{i=1}^{nb_comp} SatVar(k,i) = n \times \left\lfloor \frac{\sum_{i=1}^{nb_comp} SatVar(l,i)}{m} \right\rfloor \quad (5)$$

où $\lfloor \cdot \rfloor$ désigne la partie entière.

Dependency.

Lors de la description d'un composant, le concepteur du déploiement peut préciser si ce composant est dépendant d'un autre. Dans ce cas, il faut que les deux composants soient sur le même appareil. Soit C_k dépendant de C_l, cela se traduirait par la formule suivante :

$$\forall i \in \{1,..,nb_comp\} \quad SatVar(k,i) = 1 \implies SatVar(l,i) = 1 \quad (6)$$

5.2.3 Propriétés multi-échelles

Composants dépendants.

Les propriétés à caractère multi-échelle exprimées au moyen des mots-clés `SameValue` et `DifferentValue` portent sur plusieurs composants à la fois. Ces propriétés expriment des conditions nécessaires pour déploiement des composants, qui sont contrôlées à partir des valeurs fournies par la sonde multi-échelle concernée. Par exemple, l'expression `Ck @ SameValue Some.MS.Scale(Cl)` exprime que les composant C_k et C_l doivent être dans la même instance d'échelle de `Some.MS.Scale`. Soit $MSProbe$ la table qui associe à chaque appareil la mesure de la sonde multi-échelle, on exprime que C_k et C_l sont déployés respectivement sur A_i et A_j seulement si A_i et A_j ont la même valeur dans $MSProbe$, c'est-à-dire :

$$\forall i,j \in \{1,..,nb_app\} \quad (SatVar(k,i) = 1 \wedge SatVar(l,j) = 1) \implies (MSProbe(i) = MSProbe(j)) \quad (7)$$

5.2.4 Placement par instance d'échelle

Enfin, la présence d'une instance de composant à une certaine échelle (exprimée au moyen du mot-clé `Each`) est définie par une contrainte du même type que les précédentes, dans laquelle l'ensemble des appareils possibles est limité (et identifié à partir des valeurs mesurées par la sonde multi-échelle concernée). Par exemple, l'expression `Ck @ Each Some.MS.Scale` exprime qu'un composant C_k doit être déployé dans chaque instance d'échelle `Some.MS.Scale`. Pour cela, deux tableaux sont nécessaires : $MSProbe$, associant à chaque appareil la mesure de la sonde multi-échelle, et $MSProbeId$, listant les identifiants uniques de chaque instance d'échelle. L'expression précédente se traduirait par la contrainte (nb_inst étant le nombre d'instance d'échelle) :

$$\forall i \in \{1,..,nb_inst\} \quad \left(\sum_{\substack{j\in\{1,..,nb_app\}\\ MSProbeId(i)=MSProbe(j)}} SatVar(k,j) \right) = 1 \quad (8)$$

5.3 Résolution

Une fois toutes les contraintes définies, le solveur de contraintes cherche une solution et retourne la première trouvée.

6. EXEMPLE ET BIBLIOTHÈQUE

Dans cette section, nous commençons par présenter un exemple de code MuScADeL, puis l'application de la formalisation à ce code. Puis nous présentons notre choix du solveur de contraintes. Enfin, nous présentons une bibliothèque correspondant à la formalisation des propriétés de déploiement, ainsi que l'utilisation de cette bibliothèque.

6.1 Exemple

Nous considérons 6 composants `C1`, `C2`, `C3`, `C4`, `C5` et `C6` dont le déploiement doit satisfaire les conditions `Criter1`, `Criter2`, `Criter3`, `Criter4` et`Criter5`, ainsi que trois propriétés à caractère multi-échelle :

- `C1` doit être déployé sur un appareil qui satisfait `Criter1`, et une instance de `C1` doit être déployé dans chaque instance de `MSNetwork.Type.LAN` ;
- deux à quatre instances du composant `C2` doivent être déployées sur des appareils qui satisfont `Criter2` ;
- `C3` doit être déployé sur tout appareil du domaine étant un *smartphone* ;
- `C4` doit être déployé sur un appareil qui satisfait `Criter3` et qui doit est localisé dans la même ville que l'appareil qui héberge `C3` ;
- `C5` doit être déployé sur 7 appareils qui satisfont `Criter4` ;
- enfin, une instance de `C6` doit être déployé sur un appareil qui satisfait `Criter5` pour chaque trois instances de `C5` déployées.

Le listing 1 définit le code MuScADeL qui exprime ces propriétés.

```
Deployment {
  C1 @ Criter1, Each MSNetwork.Type.LAN;
  C2 @ 2..4, Criter2;
  C3 @ All, Device.Type.Smartphone;
  C4 @ Criter3,
        SameValue Geography.Location.City(C3);
  C5 @ 7, Criter4;
  C6 @ 1/3 C5, Criter5;
}
```

Listing 1: Exemple d'une spécification de propriétés de déploiement

6.2 Définition des matrices

La table 1a donne l'exemple de la matrice *Comp* construite à partir du code MuScADeL du listing 1. La table 1b donne l'exemple d'une matrice *Dom* extraite de l'état du domaine. Dans ces matrices, les propriétés P_1, P_2, P_3, P_4, P_5 et P_6 représentent respectivement `Criter1`, `Criter2`, `Criter3`, `Criter4`, `Criter5` et `Device.Type.Smartphone`. Il faut noter que les sondes, y compris les sondes multi-échelles (en l'occurrence, pour notre exemple, la sonde `Device`), sont utilisées pour construire la matrice *Dom*. De manière générale, les mesures prises par les sondes sur les appareils font aussi partie de l'état du domaine. Elles sont fournies sous la forme d'une table associant appareil et mesure. Ainsi, pour notre exemple, la résolution nécessite des informations de géolocalisation des appareils du domaine. C'est la sonde multi-échelle `Geography` qui détermine dans quelle ville sont localisés les appareils, et la sonde `MSNetwork` qui détermine à quel réseau local sont connectés les appareils. Elles produisent respectivement les tables 2b et 2a (ce sont des exemples).

	P_1	P_2	P_3	P_4	P_5	P_6
C_1	1	0	0	0	0	0
C_2	0	1	0	0	0	0
C_3	0	0	0	0	0	1
C_4	0	0	1	0	0	0
C_5	0	0	0	1	0	0
C_6	0	0	0	0	1	0

(a) Matrice des composants *Comp*

	P_1	P_2	P_3	P_4	P_5	P_6
A_1	1	0	1	1	1	1
A_2	1	1	0	0	1	1
A_3	1	1	1	1	0	1
A_4	1	1	1	1	0	0
A_5	1	1	1	1	0	1
A_6	1	1	0	0	1	1
A_7	0	1	1	1	1	1
A_8	1	1	1	1	0	1
A_9	0	1	1	1	1	1
A_{10}	1	1	0	1	1	1
A_{11}	1	1	1	1	0	1
A_{12}	0	1	0	0	1	1

(b) Matrice des appareils *Dom*

Table 1: Données sur les composants et les appareils

La table 3 représente une matrice d'obligation possible c'est-à-dire un plan de déploiement pour notre exemple de problème initialement défini par le listing 1 et le tableau de localisation des appareils constitué des mesures prises par les sondes `MSNetwork.Type.LAN` (cf. table 2a) et `Geography.Location.City` (cf. table 2b).

6.3 Solveur de contraintes

Afin de pouvoir choisir un solveur de contraintes qui supporte la phase de résolution, nous avons étudié et comparé un certain nombre d'outils candidats : Cream [19], Copris [18], JaCoP [11], or-tools [15], jOpt [9], et Choco [3]. Nous avons considéré le type de problème traité et le support disponible en matière d'évolution et de documentation. La table 5 présente les résultats de cette comparaison. Tous les solveurs sont compatibles avec Java (ils sont soit écrits en Java, soit interfaçables avec Java). Les acronymes CSP, COP, CP et JS correspondent respectivement à *Constraint Satisfaction Problem*, *Constraint Optimization Problem*, *Constraint Problem* et *Job Scheduling*. Notre problème étant un problème de satisfaction de contrainte (CSP), la classe de problème n'a pas été le critère de choix. Parmi ces solveurs, nous avons choisi **Choco** [3]. Nous avons déjà une certaine expertise sur cet outil, la librairie est complète et simple à utiliser.

Table 4: Constraint solvers comparison.

	Problème	Maintenance	Documentation
Choco	CSP	maintenu	complète
Copris	COP, CSP, CP	maintenu	quasi inexistante
Cream	CSP	obsolète (2008)	légère
JaCoP	CSP	maintenu	existante
jOpt	CSP, JS	maintenu	inexistante
or-tools	CSP	maintenu	quasi inexistante

Table 5: Comparaison des solveurs de contraintes

6.4 MuscadelSolving

Dans cette section, nous présentons des éléments de code Java pour la génération du plan de déploiement. Deux classes son présentées :

- Une classe `MuscadelSolving` (listing 3) et son interface `MuscadelSolvingInter` (listing 2), contenant toutes les méthodes pour l'ajout des contraintes ;
- la classe `UbiMob` (listing 12) qui présente la phase d'ajout de contraintes de l'exemple présenté dans le listing 1.

L'interface `MuscadelSolvingInter` présente les méthodes accessibles pour l'ajout des contraintes : `cardinaliteSimple`, `cardinaliteIntervalle`, `cardinaliteAll`, `ratio`, `sameValue`, `differentValue`, `each` et `resolution`[4].

```
public interface MuscadelSolvingInter {
    public void cardinaliteSimple (int cmp, int card);
    public void cardinaliteAll (int cmp);
    public void cardinaliteIntervalle(int cmp, int min, int max);
    public void ratio(int cmpP, int cmpS, int ratioP, int ratioS);
    public void sameValue(int cmp1, int cmp2, String[] sonde);
    public void differentValue(int cmp1, int cmp2, String[] sonde);
    public void each (int cmp, String[] sonde);
    public int[][] resolution() throws MuscadelSolvingExc;
}
```

Listing 2: Interface `MuscadelSolvingInter`

4. Dans le code Java, les indices de lignes et de colonnes des matrices et des tableaux commencent à 0.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}
`LAN`	orion	orion	betelgeuse	persee	cephee	orion	orion	betelgeuse	persee	cephee	persee	cephee

(a) Données de sondage de `MSNetwork.Type.LAN`

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}
`City`	Toulouse	Paris	Toulouse	Toulouse	Paris	Toulouse	Toulouse	Grenoble	Orleans	Brest	Toulouse	Orleans

(b) Données de sondage de `Geography.Location.City`

Table 2: Données de sondage des sondes multi-échelles

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}
C_1	0	0	0	0	0	1	0	1	0	1	1	0
C_2	0	0	0	0	0	0	0	0	1	1	1	1
C_3	1	0	1	0	0	1	1	0	0	0	1	0
C_4	0	0	0	0	0	0	0	0	0	0	1	0
C_5	0	0	0	1	1	0	1	1	1	1	1	0
C_6	0	0	0	0	0	0	0	0	0	1	0	1

Table 3: Matrice d'obligation *Oblig*

La classe `MuscadelSolving` contient les matrices $Comp$ et Dom, ainsi que le modèle Choco et la matrice de possibilités $SatVar$. Cette dernière est construite lors de la construction d'un objet `MuscadelSolving`, avec un appel à la méthode `pretraitement` dans le constructeur.

```
1  public class MuscadelSolving implements MuscadelSolvingInter{
2    private Model model;
3    private IntegerVariable[][] satVar;
4    private int nb_comp, nb_app nb_prop;
5    private int[][] comp, dom;
6    private ArrayList<Integer> toMaximize;

8    public MuscadelSolving (int[][] comp, int[][] dom) {
9      assert(comp.length > 0) : "Erreur␣MuscadelSolving";
10     this.model = new CPModel();
11     this.nb_comp = comp.length;
12     this.nb_app = dom.length;
13     this.nb_prop = comp[0].length;
14     this.satVar = new IntegerVariable[nb_comp][nb_app];
15     this.comp = comp;
16     this.dom = dom;
17     toMaximize = new ArrayList<Integer>();
18     pretraitement();
19   }
```

Listing 3: Classe `MuscadelSolving`

La méthode `pretraitement` construit la matrice des possibilités $SatVar$, et d'y ajouter les contraintes relatives à l'impossibilité pour l'appareil d'héberger le composant, tel que décrit par la formule (1).

```
1  private void pretraitement () {
2    int [] buffer = new int[nb_prop];
3    // Pour chaque variable, on definit le nom et le domaine
4    int[] values = {0,1};
5    for (int i = 0; i < nb_comp; i++) {
6      for (int j = 0; j < nb_app; j++) {
7        satVar[i][j] = Choco.makeIntVar("var_" + i + "_" + j, values );
8        model.addVariable(satVar[i][j]);
9      }
10   }
11   for (int i = 0; i < nb_comp; i++) {
12     for (int j = 0; j < nb_app; j++) {
13       boolean cont = true;
14       for (int k = 0; k < nb_prop; k++) {
15         if (!cont) break;
16         buffer[k] = comp[i][k] * dom[j][k];
17         cont = cont & (buffer[k] == comp[i][k]);
18       }
19       if (!cont)
20         model.addConstraint(Choco.eq(0, satVar[i][j]));
21     }
22   }
23 }
```

Listing 4: Méthode `MuscadelSolving.pretraitement`

La méthode `cardinaliteSimple` ajoute une contrainte de cardinalité simple (par exemple, dans le listing 1, à la ligne 7), tel que décrit par la formule (2).

```
1  public void cardinaliteSimple (int cmp, int card) {
2    model.addConstraint(Choco.eq(card, Choco.sum(satVar[cmp])));
3  }
```

Listing 5: Méthode `MuscadelSolving.cardinaliteSimple`

La méthode `cardinaliteIntervalle` ajoute les contraintes de cardinalité à intervalle (par exemple, dans le listing 1, à la ligne 3), tel que décrit par la formule (3). En plus de l'ajout de ces contraintes, la méthode `addConstraint` ajoute la ligne du composant à la liste des lignes de `satVar` à maximiser.

```
1  public void cardinaliteIntervalle(int cmp, int min, int max) {
2    model.addConstraint(Choco.leq(min, Choco.sum(satVar[cmp])));
3    model.addConstraint(Choco.geq(max, Choco.sum(satVar[cmp])));
4    toMaximize.add(cmp);
5  }
```

Listing 6: Méthode `MuscadelSolving.cardinaliteIntervalle`

La méthode `cardinaliteAll` ajoute les contraintes de cardinalités `All`, tel que décrit par la formule (4). Comme pour la méthode `cardinaliteIntervalle`, la méthode `cardinaliteAll` n'ajoute pas que des contraintes. Une contrainte est ajoutée pour spécifier qu'au moins un composant doit être déployé dans le domaine, et la ligne du composant dans la matrice `satVar` est ajoutée à la liste des lignes à maximiser.

```
1  public void cardinaliteAll (int cmp) {
2    model.addConstraint(Choco.leq(1, Choco.sum(satVar[cmp])));
3    toMaximize.add(cmp);
4  }
```

Listing 7: Méthode `MuscadelSolving.cardinaliteAll`

La méthode `ratio` ajoute une contrainte de ratio entre composants, tel que décrit par la formule (5). Elle prend en paramètre deux composants sur lequel se porte le ratio, le numérateur et le dénominateur.

```
1  public void  ratio (int cnum, int cdenom, int rnum, int rdenom) {
2    Constraint ratio =
3      Choco.eq(Choco.sum(satVar[cnum]),
4        Choco.mult(rnum, Choco.div(Choco.sum(satVar[cdenom]), rdenom)));
5    model.addConstraint(ratio);
6  }
```

Listing 8: Méthode `MuscadelSolving.ratio`

Les méthodes `samevalue` et `differentValue` ajoutent les contraintes relatives à la dépendance entre composants, tel que décrit dans la formule (7). Elles prennent en paramètre les composants concernés et le tableau des données du sondage (par exemple, le tableau 2b).

```
public void sameValue (int cmp1, int cmp2, String[] sonde) {
  checkValue(cmp1, cmp2, sonde, true);
}
public void differentValue(int cmp1, int cmp2, String[] sonde) {
  checkValue(cmp1, cmp2, sonde, false);
}
private void checkValue (int cmp1, int cmp2, String[] sonde,
                         boolean diff) {
  assert sonde.length == nb_app : "checkValue tab size problem !";
  for (int m1 = 0; m1 < nb_app; m1++) {
    for (int m2 = 0; m2 < nb_app; m2++) {
      if (! (diff ^ sonde[m1].equals(sonde[m2]))) continue;
      model.addConstraint(Choco.geq(1,
        Choco.plus(satVar[cmp1][m1], satVar[cmp2][m2])));
    }
  }
}
```

Listing 9: Méthode `MuscadelSolving.sameValue` et `MuscadelSolving.differentValue`

La méthode `each` ajoute les contraintes de placement d'un composant par instance d'échelle, tel que décrit par la formule (8). Elle prend en paramètre le composant concerné et un tableau des données de sondage.

```
public void each (int cmp, String[] sonde) {
  assert sonde.length == nb_app : "Each tab size problem !";
  HashMap<String, ArrayList<Integer>> id =
    new HashMap<String,ArrayList<Integer>>();
  // Construction de la map des identifiants/indexs
  for (int i = 0; i < sonde.length; i++) {
    if (id.containsKey(sonde[i]))
      id.get(sonde[i]).add(i);
    else {
      ArrayList<Integer> ids = new ArrayList<Integer>();
      ids.add(i);
      id.put(sonde[i], ids);
    }
  }
  // Ajout des contraintes
  for (String str : id.keySet()) {
    IntegerExpressionVariable add=Choco.ZERO;
    for (Integer index : id.get(str)) {
      add = Choco.plus(satVar[cmp][index], add);
    }
    Constraint check = Choco.eq(1, add);
    model.addConstraint(check);
  }
}
```

Listing 10: Méthode `MuscadelSolving.each`

La méthode `resolution` lance la résolution du solveur de contrainte. Si aucune ligne de la matrice `satVar` ne doit être maximisée, la résolution est directement lancée. Sinon, les directives de maximisation sont ajoutée puis la résolution est lancée. Par la suite, la faisabilité du problème est vérifiée : si le problème n'a pas de solution, une exception `MuscadelSolvingExc` est levée, sinon, la première solution est récupérée, et est retournée par la méthode.

```
public int[][] resolution() throws MuscadelSolvingExc {
  Solver solver = new CPSolver();
  if (toMaximize.size() == 0) {
    solver.read(model);
    solver.solve();
  } else {
    int up = nb_app*toMaximize.size();
    IntegerVariable obj = Choco.makeIntVar("max", 1, up);
    IntegerExpressionVariable add = Choco.ZERO;
    for (Iterator<Integer> it = toMaximize.iterator(); it.hasNext();) {
      Integer all = (Integer) it.next();
      add = Choco.plus(add, Choco.sum(satVar[all]));
    }
    model.addConstraint(Choco.eq(obj, add));
    solver.read(model);
    solver.maximize(solver.getVar(obj),true);
  }
  try{
    if (solver.isFeasible()) {
      int [][] resultat = new int[nb_comp][nb_app];
      for (int i = 0; i < nb_comp; i++) {
        for (int j = 0; j < nb_app; j++) {
          resultat[i][j] = solver.getVar(satVar[i][j]).getVal();
        }
      }
      return resultat;
    } else {
      throw (new MuscadelSolvingExc("Il n'y a pas de solution")); }
  } catch (NullPointerException e) {
    throw (new MuscadelSolvingExc("Il n'y a pas de solution"));}
}
```

Listing 11: Méthode `MuscadelSolving.resolution`

6.5 Utilisation de MuscadelSolving

La classe `UbiMob` contient le programme principal : construction des matrice *Dom* et *Comp* (pour l'exemple, elles sont construites à non générées par l'analyse du code MuScADeL), construction de l'objet `solv`, instance de la classe `MuscadelSolving`, appel aux différentes méthodes pour l'ajout des contraintes, résolution et affichage du résultat. Le résultat affiché est visible dans le listing 13. Ce programme représente la génération du plan de déploiement du code MuScADeL décrit dans le listing 1.

```
public class UbiMob {
  static void printOblig(int[][]oblig) { ... }
  public static void main(String[] args) {
    System.out.println("\tGeneration du plan de deploiement");
    int[][] comp = {
        {1, 0, 0, 0, 0, 0},
        {0, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 1, 0, 0, 0},
        {0, 0, 0, 1, 0, 0},
        {0, 0, 0, 0, 1, 0}};
    int[][] dom = {
        {1, 0, 1, 1, 1, 1},
        {1, 1, 0, 0, 1, 1},
        {1, 1, 1, 1, 0, 1},
        {1, 1, 1, 1, 0, 0},
        {1, 1, 1, 1, 0, 1},
        {1, 1, 0, 0, 1, 1},
        {0, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 0, 1},
        {0, 1, 1, 1, 1, 1},
        {1, 1, 0, 1, 1, 1},
        {1, 1, 1, 1, 0, 1},
        {0, 1, 0, 0, 1, 1}
    };

    MuscadelSolving solv = new MuscadelSolving (comp,dom);
    //  C1 @ Each MSNetwork.Type.LAN;
    String[] lan =
      { "orion", "orion", "betelgeuse", "persee", "cephee", "orion",
        "orion", "betelgeuse", "persee", "cephee", "persee", "cephee" };
    solv.each(0, lan);
    //  C2 @ 2..4
    solv.cardinaliteIntervalle(1,2, 4);
    //  C3 @ All
    solv.cardinaliteAll(2);
    //  C4 @ SameValue Geography.Location.City(C3);
    solv.cardinaliteSimple(3,1);
    String[] cities =
      { "Toulouse", "Paris", "Toulouse", "Toulouse", "Paris", "Toulouse",
        "Toulouse", "Grenoble", "Orleans", "Brest", "Toulouse", "Orleans" };
    solv.sameValue(2, 3, cities);
    //  C5 @ 7
    solv.cardinaliteSimple(4,7);
    //  C6 @ 1/3 C5
    solv.ratio(5, 4, 1, 3);

    try {
      int[][] oblig = solv.resolution();
      printOblig(oblig);
    } catch (MuscadelSolvingExc e) {
      System.err.println("Probleme lors de la resolution.");
      e.printStackTrace();
    }
  }
}
```

Listing 12: Classe principale `Ubimob`

```
  Generation du plan de deploiement
Oblig :
0 0 0 0 0 1 0 1 0 1 1 0
0 0 0 0 0 0 0 0 1 1 1 1
1 0 1 0 0 1 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 1 1 0 1 1 1 1 1 0
0 0 0 0 0 0 0 0 0 1 0 1
```

Listing 13: Résultat affiché

7. CONCLUSIONS ET PERSPECTIVES

Lors de travaux précédents, nous avons défini un DSL, MuScADeL, pour l'expression du déploiement autonomique multi-échelle. Dans cet article, nous présentons la formalisation de propriétés de déploiement à partir d'un modèle à base de contraintes. Les propriétés exprimées par le concepteur du déploiement dans le langage MuScADeL sont transformées en contraintes pour modéliser un plan de déploiement. En résultat, un plan de déploiement conforme à toutes les

propriétés exprimées est obtenu. Dans le cas des systèmes ambiants répartis à grande échelle, un plan de déploiement est quasiment impossible à décrire manuellement. Notre approche permet l'automatisation de la génération du plan de déploiement. La modélisation formelle des propriétés permet de garantir au concepteur de déploiement que les propriétés qu'il a exprimés sont bien respectées lors de la réalisation du déploiement. Nous avons aussi présenté une bibliothèque Java permettant la génération d'un plan de déploiement, conforme à cette formalisation, en utilisant un solveur de contraintes, Choco.

Nous avons également développé un éditeur pour MuScADeL, en utilisant les technologies Xtext et Xtend. L'éditeur est un plugin pour l'environnement de développement Eclipse, ce qui permet une facilité de prise en main et d'utilisation. L'éditeur permet de piloter directement la génération du plan de déploiement.

Actuellement, nous travaillons sur l'intergiciel de déploiement autonomique, basé sur la technologie OSGi pour la gestion de déploiement en local sur les appareils du domaine et sur des agents mobiles pour répondre au besoin d'autonomie et d'adaptation du processus.

8. REMERCIEMENTS

Ce travail fait partie d'un projet financé de l'Agence Nationale de la Recherche (ANR), le projet INCOME [5] (ANR-11-INFR-009, 2012-2015). Les auteurs remercient tous les membres du projet qui ont contribué directement ou indirectement à cet article.

9. REFERENCES

[1] Jean-Paul Arcangeli, Amel Bouzeghoub, Valérie Camps, C. Marie-Françoise Canut, Sophie Chabridon, Denis Conan, Thierry Desprats, Romain Laborde, Emmanuel Lavinal, Sébastien Leriche, Hervé Maurel, André Péninou, Chantal Taconet, and Pascale Zaraté. INCOME - Multi-scale Context Management for the Internet of Things. In Fabio Paternò, Boris E. R. de Ruyter, Panos Markopoulos, Carmen Santoro, Evert van Loenen, and Kris Luyten, editors, *Ambient Intelligence, 3rd Int. Joint Conf. AmI 2012*, volume 7683 of *Lecture Notes in Computer Science*, pages 338–347. Springer, 2012.

[2] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, Andre van der Hoek, and Alexander L. Wolf. A characterization framework for software deployment technologies. Technical report, Defense Technical Information Center (DTIC) Document, april 1998.

[3] C.H.O.C.O. Team. CHOCO : an Open Source Java Constraint Programming Library. Technical Report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[4] Alan Dearle. Software Deployment, Past, Present and Future. In Lionel C. Briand and Alexander L. Wolf, editors, *Workshop on the Future of Software Engineering (FOSE 2007)*, pages 269–284, 2007.

[5] Alan Dearle, Graham N. C. Kirby, and Andrew J. McCarthy. A framework for constraint-based deployment and autonomic management of distributed applications. In *International Conference on Autonomic Computing (ICAC'04)*, pages 300–301. IEEE Computer Society, 2004.

[6] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the grid with deployware. In *CCGRID*, pages 177–184. IEEE Computer Society, 2008.

[7] Frédéric Guidec, Nicolas Le Sommer, and Yves Mahéo. Opportunistic Software Deployment in Disconnected Mobile Ad Hoc Networks. *International Journal of Handheld Computing Research*, 1(1) :24–42, 2010.

[8] Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Int. Conf. on Software Engineering*, pages 174–183. ACM, 1999.

[9] jOpt, Java OPL Implementation. Last access : February 2014.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[11] Krzysztof Kuchcinski and Radoslaw Szymanek. JaCoP - Java Constraint Programming solver. Last access : February 2014.

[12] Christine Louberry, Philippe Roose, and Marc Dalmau. Kalimucho : Contextual Deployment for QoS Management. In Pascal Felber and Romain Rouvoy, editors, *Distributed Applications and Interoperable Systems (DAIS 2011)*, pages 43–56, 2011.

[13] Umar Manzoor and Samia Nefti. QUIET : A Methodology for Autonomous Software Deployment using Mobile Agents. *J. Network and Computer Applications*, 33(6) :696–706, 2010.

[14] Mohamed El Amine Matougui and Sébastien Leriche. Vers un environnement de déploiement autonomique. In *UbiMob'11*, volume 11, pages 57–62, 2011.

[15] or-tools, Operations Research Tools developed at Google. Last access : February 2014.

[16] Sam Rottenberg, Sébastien Leriche, Claire Lecocq, and Chantal Taconet. Vers une définition d'un système réparti multi-échelle. In *Journées francophones Mobilité et Ubiquité (UBIMOB)*. Cépaduès Editions, 2012. In French.

[17] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4) :14–23, 2009.

[18] Naoyuki Tamura. Copris : Constraint Programming in Scala. Last access : February 2014.

[19] Naoyuki Tamura. Cream : Class Library for Constraint Programming in Java. Last access : February 2014.

[20] Mahamadou Toure, Patricia Stolf, Daniel Hagimont, and Laurent Broto. Large scale deployment. In *6th Int. Conf. on Autonomic and Autonomous Systems (ICAS)*, pages 78–83. IEEE Computer Society, 2010.

[21] Sander van der Burg and Eelco Dolstra. Disnix : A Toolset for Distributed Deployment. *Science of Computer Programming*, 79 :52–69, 2014.

5. http ://anr-income.fr