Interfaces comportementales pour la reconfiguration de modèles à composants

Maverick Chardet, Hélène Coullon, Christian Pérez

▶ To cite this version:

Maverick Chardet, Hélène Coullon, Christian Pérez. Interfaces comportementales pour la reconfiguration de modèles à composants. Compas 2018 - Conférence d'informatique en Parallélisme, Architecture et Système, Jul 2018, Toulouse, France. pp.1-8. hal-01897803

HAL Id: hal-01897803
https://inria.hal.science/hal-01897803

Submitted on 17 Oct 2018

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.

Compas'2018 : Parallélisme/ Architecture / Système
ENSEEIHT - Toulouse, France, du 03 au 06 juillet 2018

Interfaces comportementales pour la reconfiguration de modèles à composants

Maverick Chardet[a], Hélène Coullon[a], Christian Perez[b]

[a] IMT Atlantique, Inria, LS2N, UBL, F-44307 Nantes, France
[b] Univ. Lyon, Inria, CNRS, ENS de Lyon, UCBL 1, LIP, Lyon, France
{maverick.chardet, helene.coullon, christian.perez}@inria.fr

Résumé
De nos jours, les infrastructures ainsi que les logiciels deviennent de plus en plus complexes et dynamiques. Du fait de cette complexité, il est important de disposer de modèles permettant d'automatiser la gestion de ces logiciels, en particulier leur reconfiguration. Les modèles à composants permettent de découpler les différentes fonctionnalités d'une application tout en exprimant explicitement leurs dépendances, les rendant particulièrement pertinents pour la reconfiguration. Ce travail est basé sur Aeolus, un modèle à composants permettant d'expliciter le cycle de vie interne de chaque composant. En revanche, lorsqu'une reconfiguration est souhaitée dans Aeolus, le développeur doit connaître le détail du cycle de vie des composants, ce qui ne respecte pas le principe de séparation des préoccupations. Nous proposons dans cet article une évolution d'Aeolus permettant la séparation des préoccupations en définissant la notion de *comportement* d'un composant.

Mots-clés : systèmes distribués, reconfiguration, modèles à composants

---

1. Introduction

De nos jours, de nombreux logiciels sont distribués, c'est-à-dire composés de plusieurs modules (ou composants) fonctionnant sur différentes machines inter-connectées (*i.e.*, sur une infrastructure distribuée). Reconfigurer un logiciel distribué consiste à le faire passer de son état courant, de configuration et de fonctionnalités, à un état souhaité différent [15]. La reconfiguration de logiciels distribués peut être souhaitée pour de nombreuses raisons : déploiement initial (rendre une application fonctionnelle depuis son état non configuré), optimisation en temps réel du rapport qualité de service/coût, évolution des services proposés (mise à jour, changement de fonctionnalités d'une application), etc. La complexité des logiciels distribués et des infrastructures sur lesquelles ils sont déployés ne cessant de s'accroître (*e.g.*, hétérogénéité, large échelle, dynamicité), il devient difficile de reconfigurer manuellement ces logiciels.
Nous nous intéressons dans cet article à l'amélioration de la séparation des préoccupations [13] entre différents acteurs d'une reconfiguration. On ne demande pas à un conducteur de savoir précisément comment fonctionne une voiture, mais simplement de connaître et maîtriser les commandes mises à sa disposition par le constructeur automobile pour la conduire. De la même manière, il est souhaitable que les concepteurs d'applications distribuées et de reconfiguration (*dev-reconf*) n'aient pas à connaître le fonctionnement détaillé des composants logiciels qu'ils assemblent et reconfigurent, mais qu'ils les manipulent seulement par des opé-

rations mises à leur disposition par les développeurs de ces modules (*dev-comp*). Nous souhaitons disposer d'un modèle qui offre l'expressivité nécessaire à la reconfiguration, qui soit non-spécifique à un type particulier de logiciel distribué, mais qui présente cependant une bonne séparation des préoccupations entre *dev-comp* et *dev-reconf*.

Les composants logiciels qui composent une application distribuée ont chacun un cycle de vie propre, c'est-à-dire que leur état, leurs dépendances et les services qu'ils fournissent varient au cours du temps. La plupart des modèles de reconfiguration existants uniformisent ce cycle de vie (*e.g.*, désinstallé, installé, en fonctionnement, suspendu). Bien que garantissant une bonne séparation des préoccupations entre les deux groupes d'acteurs, cela se fait au détriment de la performance et de la précision du modèle de reconfiguration. Aeolus [12] est un modèle à composants qui permet de modéliser de manière personnalisée le cycle de vie de chaque composant logiciel. En revanche, la séparation des préoccupations est faible car les *dev-reconf* doivent connaître les détails internes des composants qu'ils reconfigurent. Dans cet article, nous présentons une évolution du modèle Aeolus permettant de concilier la personnalisation du cycle de vie des composants et la séparation des préoccupations. Nous définissons pour cela la notion de *comportement* d'un composant et proposons un algorithme produisant une représentation externe simplifiée du composant, appelée *interface comportementale*, à destination des *dev-reconf*.

Le reste de cet article est organisé comme suit : la section 2 présente le contexte de la contribution. La section 3 présente notre modèle de reconfiguration, qui est évalué dans la section 4. Enfin, la section 5 conclut et présente quelques perspectives de ce travail.

2. Contexte

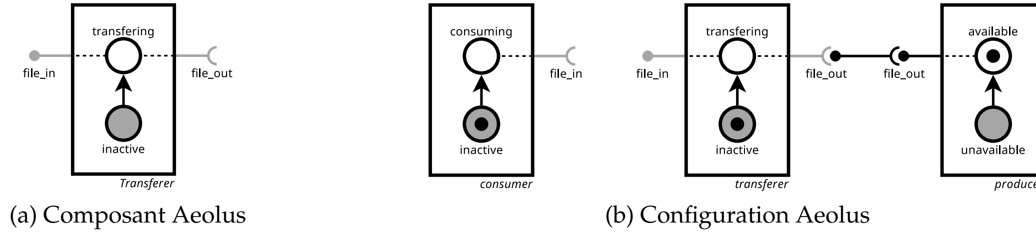
Dans cette partie, nous proposons un état de l'art sur les modèles de reconfiguration. Nous présentons ensuite le modèle Aeolus [12] sur lequel nous basons notre contribution.

2.1. État de l'art sur la reconfiguration

La reconfiguration peut être définie par le fait de faire passer un système d'un état à un autre. Elle est intéressante pour les logiciels distribués pour, par exemple, la mise à jour dynamique, la modification des services logiciels fournis ou la tolérance aux pannes. On peut considérer que le *déploiement initial* d'un logiciel distribué sur une infrastructure est une reconfiguration dans laquelle l'état source est le logiciel désinstallé. Certains outils et modèles, dits de *déploiement*, ciblent ce type particulier de reconfiguration. La plupart d'entre eux proposent en outre des mécanismes de tolérance aux pannes, d'élasticité ou d'équilibrage de charge [2, 9, 14, 8, 17].

Notre contribution s'inscrit dans un ensemble de modèles de reconfiguration plus généraux. Ils permettent de définir les reconfigurations comme des *transformations* décrivant les changements à apporter à l'état du système. Pour des raisons de concision, ce travail se concentre plus particulièrement sur la reconfiguration basés sur les *modèles à composants*. Un *composant* est une boîte noire représentant un élément logiciel. Seules ses dépendances et interfaces externes sont exposées via des *ports*. On peut assembler des composants en connectant leurs ports pour former un *assemblage*, représentant par exemple une application fonctionnelle. Deux propriétés importantes de tels modèles sont la *réutilisabilité* et la *séparation des préoccupations* [19, 4, 10, 7, 6].

Les composants d'un assemblage peuvent se trouver dans différentes phases (*e.g.*, désinstallé, en attente de configuration, en attente de lancement, suspendu, etc.). On appelle *cycle de vie* la description de ces phases et comment passer de l'une à l'autre. Dans le cas général, chaque composant a un cycle de vie qui lui est propre. Certains modèles à composants, tels que FraSCAti [18], Fractal [10] et GCM [5], permettent de personnaliser le cycle de vie des composants. Il est toutefois nécessaire de connaître leur implémentation pour comprendre comment tirer parti



(a) Composant Aeolus

(b) Configuration Aeolus

FIGURE 1 – Exemples de composant et de configuration Aeolus

de cette personnalisation, diminuant la *séparation des préoccupations*. D'autres modèles rendent ce cycle de vie visible depuis l'extérieur, mais cela se traduit par une mauvaise *séparation des préoccupations* entre *dev-comp* et *dev-reconf*. Par exemple, dans Coqcots/Pycots [11] la séparation des préoccupations est inexistante. En effet, un *dev-reconf* doit fournir de nouvelles implémentations pour les composants impliqués par la reconfiguration, de manière à minimiser les interuptions de services. Aeolus [12] considère des composants au cycle de vie personnalisable. En revanche, les instructions d'une transformation font directement référence aux états internes de ces cycles de vie, impactant également la séparation des préoccupations.

2.2. Le modèle Aeolus

Aeolus [12] est un modèle à composants de déploiement permettant de personnaliser le cycle de vie de chaque composant via une machine à états interne. Cela permet de définir de manière précise et personnalisée les étapes de déploiement et de reconfiguration des composants ainsi que de coordonner leur exécution par leurs dépendances mutuelles. Un composant Aeolus est défini par un tuple $\langle Q, q_0, T, P, D\rangle$ où Q est un ensemble d'*états*, q_0 est l'*état initial*, $T \subseteq Q \times Q$ est un ensemble de *transitions*, P est une paire d'ensembles de ports *provide* et *require* et D est une fonction indiquant pour tous les états de Q à quels ports de P ils sont reliés. La figure 1a présente un exemple de composant Aeolus, le composant *Transferer*, où $Q = \{\text{inactive}, \text{transfering}\}$, $q_0 = \text{inactive}$, $T = \{(\text{inactive} \mapsto \text{transfering})\}$, $P = \langle\{\text{file_out} \mid \text{require}\}, \{\text{file_in} \mid \text{provide}\}\rangle$, $D\,(\text{inactive}) = \emptyset$, et $D\,(\text{transfering}) = \{\text{file_out}, \text{file_in}\}$. L'état *transfering* est relié au port require *file_out* indiquant que le composant utilise un service extérieur dans cet état de son cycle de vie, et au port provide *file_in* indiquant que le composant fournit un service dans cet état.

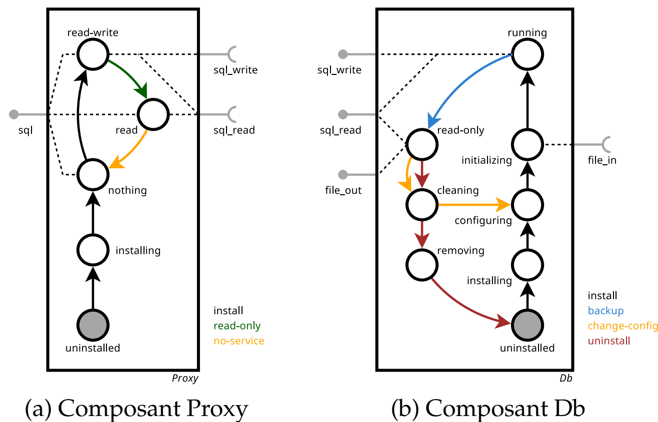
Dans Aeolus, un *assemblage* est défini par un ensemble de composants et un ensemble de *connexions* entre les ports de ces composants. Une *configuration* est un assemblage ainsi qu'un emplacement de *jeton* pour chaque composant, indiquant son *état courant*. La figure 1b présente un exemple de configuration intermédiaire lors d'un déploiement. L'assemblage est constitué de trois composants : *consumer*, *transferer* et *producer*, les deux derniers ayant leurs ports *file_out* reliés par une connexion. *consumer* et *transferer* sont dans l'état *inactive* tandis que *producer* est dans l'état *available*. Dans Aeolus, un port *provide* est dit *actif* dans une configuration si l'état courant du composant est relié à ce port. Un composant peut changer d'état courant par une transition si les ports *require* de l'état de destination sont satisfaits (*i.e.*, connectés à un port *provide* actif). Dans l'exemple, le composant *transferer* peut passer dans l'état *transfering* mais le composant *consumer* ne peut pas passer dans l'état *consuming*. Enfin, le modèle Aeolus définit cinq actions d'évolution du système : *new* et *del* pour la création et suppression de composant, *bind* et *unbind* pour la création et suppression de connexion, et *stateChange* pour le changement d'état d'un composant. Une séquence d'actions est appelée *transformation*. Un exemple de transformation pour terminer le déploiement de la figure 1b serait : (1) passer *transferer* dans l'état *transfering*, (2) connecter les deux ports *file_in*, (3) passer *consumer* dans l'état *consuming*. On remarque ici la nécessité de connaître la machine à états interne des composants.

3. Composants à interfaces comportementales

Nous présentons dans cette section une évolution d'Aeolus. Notre contribution améliore la séparation des préoccupations en introduisant la notion de *comportement*. Afin d'illustrer notre propos, nous considérons l'exemple d'une migration de base de données derrière un proxy.

3.1. Composants à comportements

Nous proposons d'ajouter aux composants Aeolus la notion de *comportement*. Le *dev-comp* définit un ou plusieurs comportements pour les composants qu'il développe, et associe chaque transition à un comportement. Intuitivement, chaque comportement désigne un ensemble de transitions successives réalisant une action voulue (*e.g.*, déploiement, mise à jour, passage dans un état dégradé). Nous représentons graphiquement chaque comportement par une couleur. Les transitions associés à ce comportement sont représentées de cette couleur.



(a) Composant Proxy (b) Composant Db

FIGURE 2 – Composants « proxy » et « base de données » avec comportements

La figure 2 représente le cycle de vie des composants *Proxy* et *Db* par une machine à états à transitions colorées. Le composant *Db* (figure 2b) a été conçu par *dev-comp* pour prendre en charge des opérations utiles à la migration. Il présente quatre comportements. Le comportement *install* installe la base de données et fait usage du port require *file_in* pour initialiser son contenu. Dans l'état *running*, les services d'écriture (*sql_write*) et de lecture (*sql_read*) dans la base sont fournis via les ports *provide* correspondants. Le comportement *backup* permet de passer dans l'état *read-only*. Dans cet état, les requêtes de lecture sont possibles, mais les requêtes en écriture sont indisponibles pour éviter toute incohérence durant la sauvegarde. De plus, le port provide *file_out* fournit la sauvegarde. Le comportement *change-config* permet quant à lui de changer la configuration de la base de données, tandis que le comportement *uninstall* la désinstalle.

Le composant *Proxy* de la figure 2a agit comme intermédiaire pour accéder à la base de donnée. Une fois lancé, il fournit le service *sql* quel que soit le niveau de service de la base de donnée (par exemple en mettant les requêtes en cache si besoin). Il présente trois comportements. Le comportement *install* installe le proxy et, si possible (ports *require* satisfaits), passe dans l'état *read-write*. Dans cet état, le composant a accès à la base de donnée en lecture (*sql_read*) et en écriture (*sql_write*). Le comportement *read-only* permet de passer dans l'état *read* dans lequel l'accès à la base de données est limité à la lecture. Le comportement *no-service* permet de passer dans l'état *nothing* dans lequel aucun accès à la base de données n'est possible.

Les règles d'évolution d'Aeolus ne sont pas impactées par l'ajout des comportements, à ceci près que chaque composant présent dans un assemblage possède un *comportement courant*. Ce dernier définit quelles transitions de la machine à états interne du composant sont autorisées.

3.2. Interfaces comportementales des composants

Les comportements d'un composant correspondent aux actions que les *dev-reconf* peuvent lui appliquer lors d'une reconfiguration. Cependant, ils n'ont pas besoin de connaître les détails de

leur exécution. Lors du déploiement du composant *Db* (figure 2b), par exemple, un *dev-reconf* n'a pas besoin d'avoir connaissance des états intermédiaires *installing*, *configuring* et *initializing* du comportement *install*. Pour cette raison, nous proposons la notion d'*interface comportementale* qui est une représentation externe simplifiée de la machine à états interne du composant. Elle contient les informations nécessaires et suffisantes à l'assemblage et au changement de comportement du composant. La figure 3 présente un assemblage dans lequel les composants *Proxy* et *Db* de la figure 2 sont représentés par leur interface comportementale. Seuls l'état initial et les états dits « stables » (ou non-transitoires) sont représentés. Chaque transition partant d'un état correspond à un changement de comportement. Si des états intermédiaires (transitoires) d'un comportement sont reliés à des ports, ces ports sont associés à la transition correspondante de l'interface comportementale.

L'interface comportementale d'un composant est générée par *dev-comp* avant publication ou par *dev-reconf* avant utilisation grâce à la fonction `GetInterface` (algorithme 1). Cette dernière stocke la liste des états à partir desquels il est possible d'appliquer un comportement dans l'ensemble *to_explore*, en commençant par l'état initial. Tant que cet ensemble n'est pas vide, elle choisit un de ses éléments et, pour chaque comportement déclenchable, appelle la fonction auxiliaire `Explore`. Cette fonction traverse chaque transition du comportement et retourne son état « stable » ainsi qu'une transition condensée. Cette dernière relie l'état initial à l'état « stable » et contient une liste de ports originellement reliés aux états intermédiaires du comportement. S'il n'a pas déjà été exploré, l'état « stable » est ajouté aux états à explorer. La transition condensée et l'état exploré sont ajoutés à l'interface comportementale. Notons que la complexité de l'algorithme est polynomiale en fonction du nombre d'états et de transitions.

3.3. Reconfiguration de composants à comportements

Une *configuration* dans notre modèle se compose d'un assemblage (composants et connexions), du comportement courant et de l'emplacement du jeton pour chaque composant. Nous adaptons les *transformations* Aeolus pour qu'elles utilisent les interfaces comportementales. Nous reprenons ainsi les actions de modification d'assemblage *new*, *del*, *bind* et *unbind*, et nous remplaçons l'action *stateChange* par deux nouvelles actions : *behaviorChange* (changement de comportement d'un composant) et *wait* (attente que le composant soit dans un état « stable »). Le *def-reconf* peut ainsi exprimer une reconfiguration en définissant une transformation, c'est-à-dire une séquence de ces actions de modification d'assemblage. Le listing 1 est un exemple de transformation pour la migration de la base de donnée de l'exemple de la figure 3. La ligne 1 change

```
1  Function GetInterface(c = ⟨Q, q0, T, P, D⟩) :
2      states ← ∅; edges ← ∅; to_explore ← {q0}
3      while to_explore ≠ ∅ do
4          extract e from to_explore
5          for ⟨o, d, b⟩ in T where o = e do
6              ⟨ns, ne⟩ ← Explore(o, d, b, c)
7              if ns ∉ states then
8                  to_explore ← to_explore ∪ {ns}
9              edges ← edges ∪ {ne}
10         states ← states ∪ {e}
11     return ⟨states, edges⟩
12 AuxFunction Explore(og,og2,bv,c = ⟨Q, q0, T, P, D⟩) :
13     ds ← og2; ports ← []
14     while ∃x : (ds, x, bv) ∈ T do
15         ports ← ports + [D (ds)]
16         ds ← x
17     return ⟨ds, ⟨og, ds, bv, ports⟩⟩
```

ALGORITHME 1 – Algorithme générant l'interface comportementale d'un composant

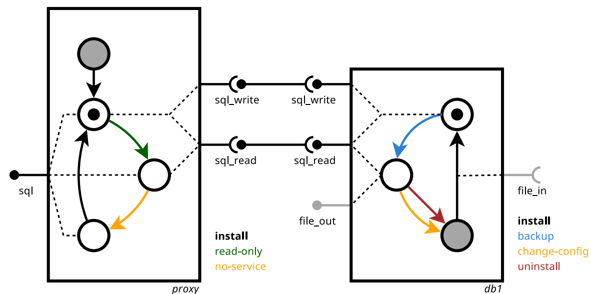


FIGURE 3 – Assemblage d'interfaces comportementales correspondantes aux composants de la figure 2

```
1   behaviorChange (proxy, read-only)
2   wait (proxy)
3   unbind (sql_write, db1, proxy)
4   new (db2 : Db)
5   new (transferer : Transferer)
6   bind (file_out, db1, transferer)
7   bind (file_in, transferer, db2)
8   behaviorChange (transferer, run)
9   behaviorChange (db2, install)
10  behaviorChange (db1, backup)
11  wait (db2)
12  behaviorChange (proxy, no-service)
13  wait (proxy)
14  unbind (sql_read, db1, proxy)
15  bind (sql_read, db2, proxy)
16  bind (sql_write, db2, proxy)
17  behaviorChange (proxy, install)
18  behaviorChange(transferer, stop)
19  wait (transferer)
20  behaviorChange(db1, uninstall)
21  wait (db1)
22  del (db1)
23  del (transferer)
```

LISTING 1 – Exemple de reconfiguration par migration de base de données sur l'assemblage de la figure 3

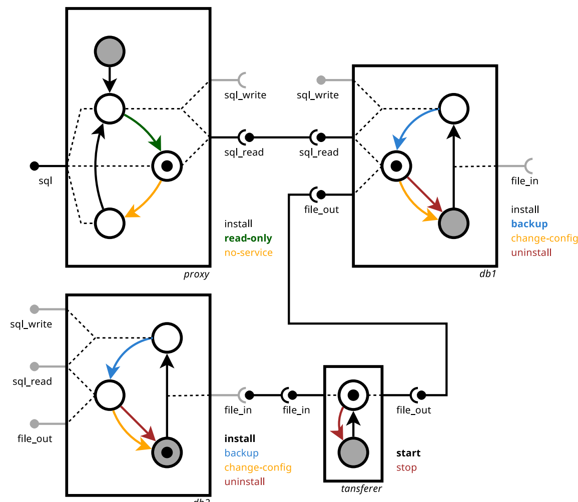


FIGURE 4 – Configuration intermédiaire lors de la migration après la ligne 10 du listing 1

le comportement du composant *proxy* en *read-only*, permettant de libérer le port *sql_write*. La ligne 2 attend que proxy ait appliqué son comportement. Les lignes de 3 à 7 déconnectent les ports *sql_write*, créent une nouvelle base *db2* ainsi qu'un composant *transferer* et les connectent. Les lignes 8 à 10 changent le comportement des deux bases et de *transferer* pour initier le transfert de données et l'installation de *db2*. La figure 4 représente la configuration après la ligne 10. La ligne 11 attend la fin de l'installation de *db2*. Les lignes 12 et 13 permettent de libérer le port *sql_read* de *proxy*. Les lignes de 14 à 16 déconnectent *db1* et connectent *db2* à *proxy*, tandis que la ligne 17 permet à *proxy* de réutiliser les ports *sql_read* et *sql_write*. Enfin, les lignes de 18 à 23 préparent *db1* et *transferer* à leur suppression et les supprime.

4. Évaluation

Notre évaluation consiste à comparer notre transformation (listing 1) à la même transformation exprimée dans Aeolus (listing 2). Les deux transformations font 23 lignes. En effet, les instructions *changeBehavior* de notre modèle correspondent à plusieurs instructions Aeolus de type *stateChange*. Par exemple, la ligne 9 du listing 1 équivaut aux lignes 8, 9, 11 et 12 du listing 2, faisant passer le composant *db2* successivement dans les états *installing*, *configuring*, *initializing* et *running* (figure 5). En revanche, notre notion de comportement demande une forme d'asynchronisme, ce qui impose l'ajout d'instructions *wait*. La taille de la transformation n'est donc pas réduite dans notre exemple. Là n'était toutefois pas le but recherché. Notons que l'introduction d'asynchronisme ouvre la porte à une forme de parallélisation de la reconfiguration.

Sur la séparation des préoccupations, nous remarquons qu'alors que les treize lignes rouges de la transformation Aeolus (listing 2) font référence aux états internes des composants, aucune référence n'y est faite dans la transformation permise par notre modèle (listing 1). Les interfaces comportementales simplifient la représentation externe des composants tout en donnant les informations nécessaires aux *dev-reconf*. Notre contribution améliore donc la séparation des préoccupations entre *dev-comp* et *dev-reconf*, ce qui était notre objectif. À notre connaissance, aucun autre modèle avec un cyle de vie personnalisable pour les composants ne permet à un *dev-reconf* de l'exploiter tout en conservant une bonne séparation des préoccupations.

On peut objecter que notre notion de comportement représente un travail supplémentaire pour le *dev-comp*, qui doit définir des actions de haut niveau en associant un comportement à chaque

```
1   stateChange (proxy, read)
2   unbind (sql_write, db1, proxy)
3   new (db2 : Db)
4   new (transferer : Transferer)
5   bind (file_out, db1, transferer)
6   bind (file_in, transferer, db2)
7   stateChange (transferer, running)
8   stateChange (db2, installing)
9   stateChange (db2, configuring)
10  stateChange (db1, read-only)
11  stateChange (db2, initializing)
12  stateChange (db2, running)
13  stateChange (proxy, nothing)
14  unbind (sql_read, db1, proxy)
15  bind (sql_read, db2, proxy)
16  bind (sql_write, db2, proxy)
17  stateChange (proxy, read-write)
18  stateChange(transferer, idle)
19  stateChange(db1, cleaning)
20  stateChange(db1, removing)
21  stateChange(db1, uninstalled)
22  del (db1)
23  del (transferer)
```

LISTING 2 – Exemple de reconfiguration Aeolus par migration de base de données sur l'assemblage de la figure 3

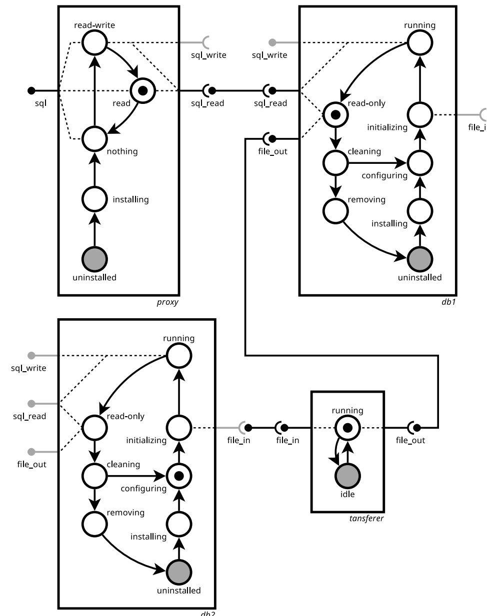


FIGURE 5 – Configuration intermédiaire lors de la migration après la ligne 10 du listing 2

transition. Cependant, il s'agit de déplacer la charge de travail du *dev-reconf* au *dev-comp*. Le travail demandé au *dev-reconf* est ainsi plus adapté à son rôle. En effet, ce dernier n'est pas censé connaître le fonctionnement du composant. Il est toutefois envisagé dans de futurs travaux de réaliser un ensemble de *templates* qui permettront de simplifier ce processus en proposant une base pour le développement d'un composant d'une famille courante (*e.g.*, base de données).

Enfin, il existe d'autres modèles que les machines à états à transitions colorées pour représenter les comportements, tels que les machines à états d'UML [1]. Ces dernières permettent de définir une hiérarchie, ce qui pourrait remplacer l'utilisation des transitions colorées ainsi que l'algorithme 1. En revanche, la représentation serait moins intuitive pour le public visé.

5. Conclusion et perspectives

Nous avons présenté dans cet article une évolution du modèle Aeolus permettant aux développeurs de composants logiciels de définir des *comportement* pour chaque composant, c'est-à-dire une succession d'actions à effectuer sur un composant pour arriver dans un état donné. Il est alors possible de fournir aux développeurs de reconfiguration l'*interface comportementale* d'un composant, une représentation simplifiée comportant uniquement les informations utiles au développeur de reconfiguration. Cela conduit à une amélioration de la séparation des préoccupations entre développeurs de composants et développeurs de reconfiguration.

Il serait intéressant dans nos futurs travaux de formaliser le modèle et sa sémantique opérationnelle. Cela permettrait de réaliser des preuves, par exemple de correction et de terminaison d'une transformation. Une piste envisagée est la transformation d'un assemblage de notre modèle vers un formalisme de machines à états tels que les machines à états d'UML [1] ou les réseaux de Petri [16] pour traitement par un vérificateur de modèles. La prise en charge du parallélisme d'actions à effectuer dans un composant lors d'une reconfiguration serait aussi un domaine à explorer. En effet, le modèle offrirait vraissemblablement de meilleures performances, tout en conservant sa simplicité d'utilisation grâce aux interfaces comportementales.

Enfin, nous visons à utiliser ce modèle sur des cas concrets, par exemple la reconfiguration d'une version décentralisée d'*OpenStack* dans un contexte *Fog Computing* [3].

Bibliographie

1. About the Unified Modeling Language Specification. – `https://www.omg.org/spec/UML/About-UML/`.
2. Juju. – `https://www.ubuntu.com/cloud/juju`.
3. The Discovery Initiative. – `http://beyondtheclouds.github.io/`.
4. About the CORBA Component Model Specification Version 4.0. – `https://www.omg.org/spec/CCM/`, 2006.
5. Baude (F.), Caromel (D.), Dalmasso (C.), Danelutto (M.), Getov (V.), Henrio (L.) et Pérez (C.). – GCM : a grid extension to Fractal for autonomous distributed components. *annals of telecommunications - annales des télécommunications*, vol. 64, n1-2, feb 2009, pp. 5–24.
6. Bigot (J.), Hou (Z.), Perez (C.) et Pichon (V.). – A Low Level Component Model Enabling Performance Portability of HPC Applications. – In *2012 SC Companion : High Performance Computing, Networking Storage and Analysis*, pp. 701–710. IEEE, nov 2012.
7. Bigot (J.) et Pérez (C.). – High Performance Composition Operators in Component Models. vol. 20, 2011, pp. 182 – 201.
8. Boujbel (R.). – *Déploiement de systèmes répartis multi-échelles : processus, langage et outils intergiciels*. – Thèse de PhD, Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier), 2015.
9. Brewer (E. A.) et A. (E.). – Kubernetes and the path to cloud native. – In *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC '15*, pp. 167–167, New York, New York, USA, 2015. ACM Press.
10. Bruneton (E.), Coupaye (T.), Leclercq (M.), Quéma (V.) et Stefani (J.-B.). – The FRACTAL component model and its support in Java. *Software : Practice and Experience*, vol. 36, n11-12, sep 2006, pp. 1257–1284.
11. Buisson (J.), Dagnat (F.), Leroux (E.) et Martinez (S.). – Safe reconfiguration of Coqcots and Pycots components. *Journal of Systems and Software*, vol. 122, dec 2016, pp. 430–444.
12. Di Cosmo (R.), Mauro (J.), Zacchiroli (S.) et Zavattaro (G.). – Aeolus : a Component Model for the Cloud. *Information and Computation*, 2014, pp. 100–121.
13. Ernst (E.). – Separation of concerns. – In *Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), Boston, MA, USA*, 2003.
14. Flissi (A.), Dubus (J.), Dolet (N.) et Merle (P.). – Deploying on the Grid with DeployWare. – In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pp. 177–184. IEEE, may 2008.
15. Kramer (J.) et Magee (J.). – The evolving philosophers problem : dynamic change management. *IEEE Transactions on Software Engineering*, vol. 16, n11, 1990, pp. 1293–1306.
16. Murata (T.). – Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, vol. 77, n4, apr 1989, pp. 541–580.
17. Paraiso (F.). – *socloud : distributed Multi-Cloud Platform for deploying, executing and managing distributed applications*. – Theses, Université des Sciences et Technologie de Lille - Lille I, 2014.
18. Seinturier (L.), Merle (P.), Fournier (D.), Dolet (N.), Schiavoni (V.) et Stefani (J.-B.). – Reconfigurable SCA Applications with the FraSCAti Platform. – In *2009 IEEE International Conference on Services Computing*, pp. 268–275. IEEE, 2009.
19. Szyperski (C.), Gruntz (D.) et Murer (S.). – *Component software : beyond object-oriented programming*. – ACM Press, 2002, *Component software series*.