Exécution décentralisée de workflow par composition moléculaire

Hector Fernandez

RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Exécution décentralisée de workflow par composition moléculaire

Héctor Fernández (*hector.fernandez@inria.fr*)

INRIA Rennes - Bretagne Atlantique

Résumé
Avec le développement de l'Internet des services, composer des services distribués faiblement couplés dynamiquement est devenu le nouveau *challenge* du calcul à large échelle. Alors que la composition de services est devenue un élément clef des plates-formes orientées service, les systèmes de composition de services suivent pour la plupart une approche centralisée, entraînant des problèmes à la fois techniques (goulot d'étranglement, tolérance aux pannes) mais aussi sociétaux ou environnementaux (protection de la vie privée, consommation d'énergie . . .). Conjointement, des langages de description de *workflow*, tels BPEL, ne peuvent s'exécuter que dans des environnements statiques et centralisés. Il devient important de promouvoir des systèmes de composition de services permettant la coordination de ces services de façon décentralisée et autonome. Récemment, s'inspirer des processus naturels s'est avéré une piste prometteuse pour la coordination de services autonomes. Dans cet article, nous nous appuyons sur une analogie inspirée par la nature basée sur la *composition moléculaire*. Selon cette analogie, les services sont des molécules qui flottent dans une solution chimique. La coordination de ces services est effectuée par un ensemble de réactions entre ces molécules exprimant l'exécution décentralisée d'un *workflow*. Nous montrons comment combiner des règles de réactions (les règles étant elles-même des molécules) pour traiter une large variété de schéma de *workflow*. Dans cette voie, nous proposons une extension de la notion de calcul chimique pour l'exécution décentralisée et dynamique de *workflow*.

Mots-clés : Composition des services, exécution paralléle, coordination décentralisée.

1. Introduction

Les architectures orientées services (SOA, pour *Service-Oriented Architectures*) se sont développés autour de l'idée de la composition de services autonomes faiblement couplés [1]. La combinaison de services permet de construire des applications plus complexes, notamment par leur composition temporelle sous forme de *workflow*, c'est-à-dire par la spécification de l'enchaînement temporelle des différents services mis-en-œuvre. Les systèmes actuels de gerstion de ces *workflows* s'appuient sur un coordinateur centralisée connaissant l'ensemble des informations de contrôle de flux et de données du workflow, posant des problèmes de passage à l'échelle, de fiabilité, mais aussi de protection de la vie privée ou de consommation d'énergie [2]. Il apparait ainsi nécessaire de développer des architectures alternatives décentralisées pour cette coordination.
Dernièrement, des métaphores naturelles ont étés mises en avant pour le développement de nouvelles approches pour la coordination de services [3]. Dans cet article, nous proposons une nouvelle analogie avec la composition moléculaire pour l'exécution décentralisée d'une grande variété de schémas de workflow, ou *workflow patterns* [1]. Selon cette analogie, les interactions entre les services à l'exécution sont modélisés par des réactions entre des molécules. Notre approche se fonde sur le modèle de programmation chimique, dans lequel un programme est vu comme une série de réactions parallèles, distribuées et autonomes entre les molécules d'une solution chimique. Ce modèle offre des propriétés intéressantes pour la coordination dans des environnements dynamiques et décentralisés [4] et a été mis en œuvre par le langage HOCL [5] sur lequel nous nous appuyons. A partir d'un cadre architectural proposé précédemment [6], nous présentons ici comment résoudre de façon décentralisée une large variété de *workflow patterns* par la composition de molécules exprimant les règles d'interactions des services.
Le paradigme chimique ainsi que notre cadre architectural est décrit dans la section 2. Nous présentons notre modèle de coordination décentralisée dans la section 3, illustré ensuite dans la section 4. Enfin, avant de conclure, nous présentons les travaux connexes à cette approche dans la section 5.

1. En français, on peut traduire par "schéma" ou "patron de flux de contrôle". Toutefois, pour des raisons d'usage et de lisibilité, on emploiera le plus souvent "workflow pattern".

2. La métaphore chimique

Des analogies avec la nature, et plus particulièrement bio-chimiques, ont été récemment proposées dans la construction des modèles de programmation du futur Internet des services [3]. Initialement développé comme un langage naturellement parallèle, le paradigme de programmation chimique exhibe des propriétés recherchées dans les plates-formes de services émergentes.

2.1. Le paradigme de programmation chimique

Selon la métaphore chimique, des molécules (données) flottent dans une solution, et réagissent selon des règles de réactions (programmes) produisant de nouvelles molécules (résultats). Ces réactions ont lieu de façon implicitement parallèle, dans un ordre indéterminé jusqu'à l'état d'*inertie*, dans lequel plus aucune réaction n'est possible. Formellement, les molécules forment un multi-ensemble réécrit par les règles représentant les réactions [7].

Plus récemment, un langage chimique d'ordre supérieur (HOCL, pour *Higher Order Chemical Language*) [5], a été proposé. Dans HOCL, chaque entité (donnée ou règle) est une molécule. Un programme est un multi-ensemble d'atomes, noté $A_1, A_2, \ldots, A_n$, "," étant l'opérateur de construction associatif et commutatif de molécules composites. Un atome peut être une constante (entier, booléen ...), une règle de réaction, une sous-solution (notée $\langle M_i \rangle$) ou un tuple (noté $A_1 : A_2 : \ldots : A_n$), composées de n atomes. Une réaction implique une règle **one** P **by** M **if** V et une molécule N satisfaisant le schéma P et la condition V. La réaction consomme la règle et N et produit une molécule M. Cette règle est *one-shot* : elle disparait après sa réaction. Sa variante **replace** P **by** M **if** V est $n - shots$: elle n'est pas consommée par les réactions et reste dans la solution pour d'autres réactions éventuelles. Il est possible de nommer les molécules. Considérons le programme chimique qui calcul la valeur maximal d'un multi-ensemble d'entiers : **let** $max =$ **replace** x, y **by** x **if** $x \geq y$ **in** $\langle 2, 3, 5, 8, 9, max \rangle$. La règle max réagit avec deux entiers x et y tel que $x \geq y$ et les remplace par x seul. Afin d'extraire le résultat de la solution, nous avons besoin de supprimer la règle max une fois que la solution ne contient que max et la plus grande valeur. Cela introduit un besoin de séquentialité : on doit attendre l'inertie de la solution d'entiers. Cela se fait par l'introduction de sous-solution. Par définition, pour accéder au contenu d'une sous-solution, il faut attendre son inertie. Dans notre exemple, cela passe par l'encapsulation de la solution d'entiers, et de l'introduction d'une règle d'ordre supérieur supprimant la règle max et extrayant le résultat : **one** $\langle max = m, \omega \rangle$ **by** $\omega \rangle$. A l'exécution, les entiers réagissent d'abord avec la règle max, produisant l'état intermédiaire. Une fois que l'inertie est atteinte au sein de la sous-solution, la règle d'extraction *one-shot* est exécutée :

$$\langle \langle 2, 3, 5, 8, 9, max \rangle, \textbf{ one } \langle max = m, \omega \rangle \textbf{ by } \omega \rangle \rightarrow^* \langle \langle 9, max \rangle, \textbf{ one } \langle max = m, \omega \rangle \textbf{ by } \omega \rangle \rightarrow \langle 9 \rangle$$

2.2. Une architecture pour la coordination chimique

Nous avons proposé un *framework* architectural pour la coordination décentralisée de workflow basé sur HOCL [6]. Cette architecture a pour objectif de permettre à plusieurs processus (services) de se coordonner selon le modèle chimique. Le workflow à exécuter est exprimé comme une solution chimique contenant toutes les molécules représentant l'information (flux de données et de contrôle) nécessaire à cette exécution. Le multi-ensemble devient alors un espace partagé par les services. Chaque service exécutera une partie de la coordination en communiquant avec les autres services en lisant et (ré)écrivant ce multi-ensemble. Pour ce faire, les services sont encapsulés dans des web services chimiques (ou ChWS, pour *Chemical Web Services*), intégrant un interpréteur HOCL jouant le rôle de moteur de workflow local.

3. Exécution décentralisée par composition Moléculaire

Fondée sur l'architecture présentée dans la Section 2.2, nous montrons maintenant comment exprimer et exécuter une grande variété de *workflow patterns* par la composition de molécules distribuées, représentant les ChWS, les données échangées, mais aussi les règles de réactions permettant leur coordination. Nous distinguons deux types de molécules, celles qui décrivent un workflow spécifique de celle, **génériques**, qui vont permettre l'éxécution décentralisée de n'importe quelle workflow. Ces dernières constituant une partie du moteur de workflow. La figure 1 représente, de gauche à droite, un workflow composé par quatre services, sa représentation chimique et la composition moléculaire de son exécution.

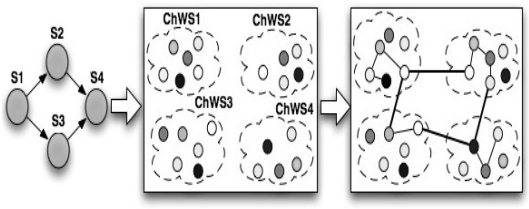


FIGURE 1 – Workflow et sa composition moléculaire.

```
2.01  ChWS1 :⟨Dest :"ChWS2",Dest :"ChWS3"⟩,
2.02  ChWS2 :⟨Dest :"ChWS4", replace Result :ChWS1 :R1
2.03          by Call :ChWS2 :(R1) :result⟩,
2.04  ChWS3 :⟨Dest :"ChWS4", replace Result :ChWS1 :R1
2.05          by Call :ChWS3 :(R1) :result⟩,
2.06  ChWS4 :⟨replace Result :ChWS2 :R2, Result :ChWS3 :R3
2.07          by Call :ChWS4 :(R2) :result⟩
```

FIGURE 2 – Représentation d'un workflow.

3.1. Représentation chimique d'un workflow

La solution principale représentant un workflow est composée d'autant de sous-solutions que nous avons de services dans le *workflow*. Comme illustré dans la figure 2, chaque sous-solution représente un service avec ses données et ses dépendances. Formellement, un ChWS est représenté par une molécule de la forme $ChWSi : \langle \ldots \rangle$. Dans l'exemple, il est composé de quatre services S_1, S_2, S_3 et S_4. Après que S_1 termine, S_2 et S_3 peuvent être invoqués en paralléle. Une fois que S_2 et S_3 terminent à leur tour, S_4 peut être invoqué. Les dépendances entre les ChWS sont exprimées par des molécules de la forme $Dest : "ChWSi"$. Par exemple, nous pouvons voir dans la sous-solution du $ChWS1$ que le résultat de l'invocation du service qu'il encapsule doit être envoyé vers le $ChWS_2$ et le $ChWS_3$ (voir la ligne 2.01). Les ChWS2 et ChWS3 sont similaires et ont une dépendance de donnée : l'invocation du service S_2 ne peut être déclenchée que lorsqu'une molécule de type *Result :ChWS1 :R1* qui contient le résultat S_1 est présent dans la sous-solution $ChWS_2$. L'invocation possible d'un service est exprimé par une molécule de la forme *Call :ChWSi :(Rj) :result* présente dans la sous-solution du service i, où R_j est le résultat d'un service S_j précédent et où *result* est le résultat de cette invocation.
Ce fragment de code HOCL constitue la représentation chimique d'un *workflow* qui sera interprété par un moteur de *workflow* contenant un ensemble de règles générique permettant de l'exécuter de façon décentralisée, que nous détaillons maintenant.

3.2. Règles chimiques pour l'exécution distribuée

Rappelons que tout est molécule : services, données, messages ou les règles de réaction elles-même. Considérons d'abord trois exemples de ces règles *génériques* présentées dans la Figure 3. La règle *invoke-Serv* réagit avec un tuple *Call :ChWSi :params :Resulti* (à l'intérieur de la sous-solution *ChWSi*) qui va réaliser l'invocation au service web S_i et produire le résultat du service. Cette règle est présente dans tous les ChWS, et parfois implicite dans la suite [2] La molécule *Invoke :value* indique si l'invocation peut avoir lieu dans le cas d'une invocation conditionnelle (on verra des exemples dans la suite), $value$ pouvant prendre la valeur 0 ou 1. La règle *addResult* prépare une donnée à son transfert en l'encapsulant dans une molécule représentant un message qui réagira à son tour avec la règle *passInfo* qui réalise le transfert du message de la solution d'un service source vers la solution d'un service destination. Ces règles sont des briques de base qui vont être comlposées avec d'autres règles permettant la coordination des services dans des *pattern* plus complexes.

3.3. Coordination décentralisée des workflow patterns

Nous présentons maintenant les compositions de molécules permettant l'exécution décentralisée des principaux *workflow patterns*.

3.3.1. Parallélisme

Le *pattern parallélisme*, illustré par la figure 4, représente la division d'une branche (ou flux) en plusieurs s'exécutant en parallèle. Il s'agit donc pour un ChWS, d'envoyer le résultat en parallèle vers différents services initiant des activités parallèles. Du point de vue chimique, nous utilisons uniquement la règle *passInfo* (*A*), décrite précédemment qui va réagir autant de fois qu'il y a de ChWS destination.

2. L'implémentation d'une telle molécule dans une plate-forme réelle, que nous ne détaillons pas ici, suppose l'interface du moteur chimique de *workflow* vers le réseau et le service web invoqué .

```
3.01  let invokeServ = replace ChWSi :⟨Call :Si :params :result, ω ⟩, Invoke :value
3.02                   by ChWSi :⟨ResultSi :result, ω ⟩      if ( value == 1)
3.03  let addResult = replace ChWSi :⟨Result :ChWSi :Resulti, Dest :"ChWSj"⟩
3.04                   by ChWSi :⟨Pass :ChWSj :⟨ResultSi :result⟩ ⟩
3.05  let passInfo = replace ChWSj :⟨Pass :idChWS :⟨ ω1 ⟩ ⟩, ChWSi :⟨ ω2 ⟩
3.06                   by ChWSi :⟨ ω1, ω2 ⟩      if (idChWS == ChWSi)
```

FIGURE 3 – Règles génériques de base.

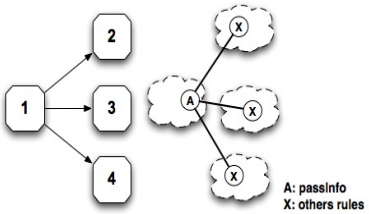
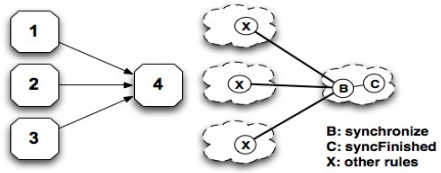


FIGURE 4 – Parallélisme

3.3.2. Synchronisation

Illustrée par la figure 5, la synchronisation réalise la fusion de plusieurs branches. Du point de vue chimique, cela revient à compter, par la règle *synchronize* (B) le nombre de résultats de la forme*Completed :ChWSi :Resulti* reçu par le ChWS destination. La molécule *In :iterator* maintient ce nombre, et *Start_Invocation* :⟨ ω ⟩ garde toutes las molécules *Completed :ChWSi :Resulti* reçues. Lorsque ce nombre est atteint, la réaction *syncFinished* (C) réagit pour produire la molécule *Invoke :value* (voir la ligne 4.06), qui indiquera à la molécule *invokeServ* que l'invocation peut avoir lieu.

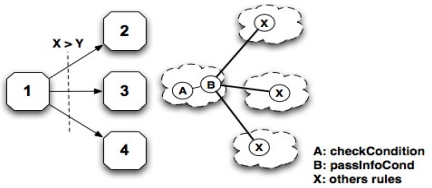



```
4.01  let synchronize = replace In :iterator, Completed :ChWSi :Resulti,
4.02                          Start_Invocation : ⟨ ω ⟩
4.03                  by Start_Invocation :⟨
4.04                          Completed :ChWSi :Resulti, ω ⟩, In :(iterator-1)
4.05                  if (iterator ≥ 1)
4.06  let syncFinished = replace-one In :iterator
4.07                  by Invoke :1      if (iterator == 0)
```

FIGURE 5 – Règles chimiques - Synchronisation

3.3.3. Choix exclusif

Comme illustré par la figure 6, un choix exclusif permet de choisir dynamiquement une branche parmi plusieurs. Du point de vue du modèle chimique, cela consiste à envoyer des molécules depuis un ChWS vers un autre uniquement lorsqu'une condition est satisfaite, ce qui est exprimé par une molécule de la forme *Condition_Pass :conditionValue*. Cette molécule réagira avec la molécule-règle *passInfoCond* (B) uniquement si sa deuxième partie est à 1.



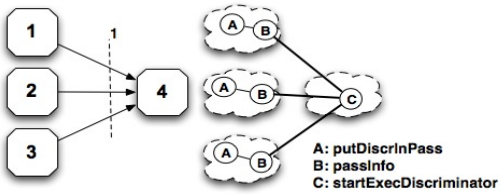

```
5.01  let passInfoCond = replace ChWSj :⟨Pass :idChWS :⟨ ω1 ⟩ ⟩, ChWSi :⟨ ω2 ⟩,
5.02                          Condition_Pass :conditionValue
5.03                  by ChWSi :⟨ ω1, ω2 ⟩, Condition_Pass :value
5.04                  if( (conditionValue==1) && (idChWS == ChWSi) )
```

FIGURE 6 – Règles chimiques - Choix exclusif

3.3.4. Discriminateur

Illustré par la figure 7, le *pattern discriminateur* représente un point dans le *workflow* où un ChWS qui a plusieurs flux en entrée ne sera déclenché qu'une seule fois, les flux suivants étant ignorés. Son implémentation chimique consiste à attendre sur ce ChWS une molécule de la forme *Discriminator :value*,

qui ajoutée dans les ChWS sources par la règle*putDiscrInPass* (A) au message transféré par la règle *passInfo* (B) déjà évoquée, indiquant que seul le premier résultat reçu doit donner lieu à invocation (voir les lignes 6.01 à 6.03). Sur le ChWS destinataire, une fois cette molécule reçue, une molécule *Invoke :1* est crée par la réaction de la règles *startExecDiscriminator* permettant l'invocation future du service (voir les lignes 6.04 à 6.05).

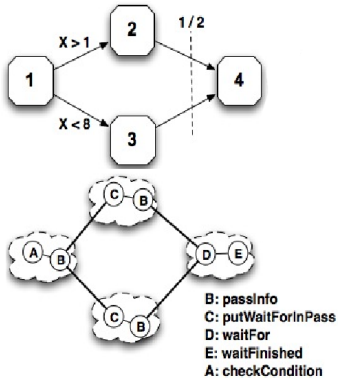



```
6.01  let putDiscrInPass = replace Dest :"ChWSj", Result :ChWSi :Resulti
6.02                       by Pass :ChWSj :⟨Result :ChWSi :Resulti,
6.03                          Discriminator :Yes⟩
6.04  let startExecDiscriminator = replace-one Discriminator :value
6.05                       by Invoke :1        if (value == Yes)
```

FIGURE 7 – Règles chimiques - Discriminateur

3.3.5. Fusion Synchronisée

La fusion synchronisée est un point où **un ou plusieurs** flux d'entrée d'un ChWS peuvent être actifs en fonction d'un choix effectué précédemment, la synchronisation n'étant nécessaire que lorsque plusieurs de ces flux sont actifs. Par exemple, sur la partie gauche de la figure 8, en fonction de la valeur de X, un ou plusieurs de ses flux de sorties sont activés. En conséquence, un ou plusieurs des flux d'entrée du $ChWS_4$ peuvent être actifs. Son implémentation chimique consiste à envoyer depuis le ChWS initiateur du *pattern* (le 1 sur la figure 8, à travers une molécule notée A spécifique du *workflow*, et la règle *passInfoCond* (B)), une molécule*WaitFor :number*, qui donne le nombre de molécules *Completed :ChWSi :Resulti* à attendre avant de s'exécuter aux services intermédiaires activés (2, 3, ou 2 et 3 sur la figure) qui vont intégrer la molécule *WaitFor :number* à la transmission de leur résultat au ChWS destinataire (le 4 sur la figure), à travers la réaction de la règle *putWaitForInPass* (C). Sur ce dernier, la règle *waitFor* (D) attend donc les résultats nécessaires et la règle *waitFinished* (E) rèagit et produit une nouvelle molécule *Invoke :1* qui permettra l'invocation future du service encapsulé par le $ChWS_4$.




```
7.01  let putWaitForInPass = replace Dest :"ChWSj", Result :ChWSi :Resulti,
7.02                          WaitFor :number
7.03                       by Pass :ChWSj :⟨Completed :ChWSi :Resulti,
7.04                          WaitFor :number⟩,WaitFor :number,
7.05                          Result :ChWSi :Resulti
7.06  let waitFor = replace WaitFor :number, Completed :ChWSi :Resulti,
7.07                        Finish_Wait :⟨ ω ⟩
7.08                     by Finish_Wait :⟨Completed :ChWSi :Resulti, ω ⟩,
7.09                        WaitFor :(number-1)
7.10                     if (number ≥ 1)
7.11  let waitFinished = replace-one WaitFor :number
7.12                     by Invoke :1        if (number == 0)
```

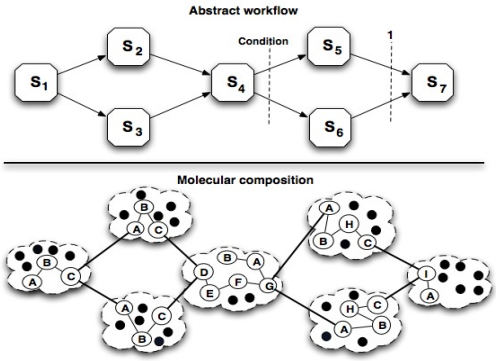
FIGURE 8 – Règles chimiques - Fusion synchronisée

Nous ne détaillons pas d'autres *pattern* (*fusion simple*, *choix multiple* ...) pour des raisons d'espace.

4. Exemple d'exécution

Sur la figure 9, on montre en partie haute sept WS chimiques appliquant les schémas *parallèlisme*, *synchronisation*, *choix exclusif*, *discriminateur*. Sur la partie basse, on peut voir le graphe de la composition moléculaire du *workflow*. Nous nous réferons à ce graphe tout au long de cette section. Les liens dessinés

entre les molécules expriment une relation entre ces molécules. Cette relation peut être de cause à effet, la réaction de l'une provoquant la réaction de l'autre, ou l'une produisant une information utilisée par l'autre lors de sa réaction. Nous détaillons l'exécution de ce *workflow* pas à pas dans les figures 10 et 11. Initialement, la représentation chimique de ce *workflow* est illustré par la figure 9.




```
8.01 ChWS1 :⟨Dest :"ChWS2", Dest :"ChWS3", passInfo, addResult, invokeServ⟩,
8.02 ChWS2 :⟨Dest :"ChWS4", passInfo, invokeServ, addResult⟩,
8.03 ChWS3 :⟨Dest :"ChWS4", passInfo, invokeServ, addResult⟩,
8.04 ChWS4 :⟨In :2, passInfoCond, synchronize, syncFinished,
8.05       Start_invocation :⟨ ω ⟩, invokeServ, addResult,
8.06       (replace-one condition_param by Condition_Pass :1,
8.07             Dest :"ChWS5", Dest :"ChWS6" if ( condition ))),
8.08 ChWS5 :⟨Dest :"ChWS7", invokeServ, passInfo, putDiscrInPass⟩,
8.09 ChWS6 :⟨Dest :"ChWS7", invokeServ , passInfo, putDiscrInPass⟩,
8.10 ChWS7 :⟨invokeServ, startExecDiscriminator⟩
```

FIGURE 9 – Definition du workflow.

Considérons dans un premier temps le bloc constitué de *ChWS1*, *ChWS2*, *ChWS3*, *ChWS4*. Par la réaction de sa molécule *invokeServ* (notée A sur la figure), *ChWS1* appelle le service qu'il encapsule et crée ainsi une molécule de la forme *Result :ChWS1 :R1*. Cette nouvelle molécule, combinée avec les molécules *addResult* et les deux molécules de type *Dest :ChWSi*, vont créer les molécules encapsulant cette molécule résultat qui vont être transférées vers les ChWS destinataires. La molécule *passInfo* déclenche ce transfert. (Voir les lignes 9.02 à 9.17.)
Une fois l'information reçue par *ChWS2* et *ChWS3*, la règle *invokeServ* (*A*) est déclenchée pour produire deux nouvelles molécules*Result :ChWS2 :R2* et *Result :ChWS3 :R3* qui seront transférés à *ChWS4* (voir les lignes 9.19 à 9.39). Ainsi, *ChWS4* attend la fin des services encapsulés par *ChWS2* et *ChWS3* pour se lancer à son tour. Cela se fait à travers la composition des molécules *synchronize* (*D*) et *syncFinished* (*E*). Une molécule *Result :ChWS4 :R4* est alors générée (voir la ligne 10.04). Ensuite, *ChWS4* déclenche la réaction avec la règle (*F*) en charge de tester la condition (non détaillée ici car dépendant de chaque condition) et de donner le résultat du test à la molécule dont la réaction va envoyer (ou non) le résultat de l'invocation du service encapsulé par *ChWS4* aux services destinataires (*ChWS5* et *ChWS6*), réalisant ainsi le *pattern* de *choix exclusif*. Considérons maintenant que le résultat de l'invocation des services à l'intérieur de *ChWS5* et *ChWS6* (ou de l'un d'entre eux suivant le résultat du *pattern* précédent), a été produit (voir les lignes 10.11 à 11.11). Les règles *putDiscrInPass* (*H*) sont déclenchés par les moteurs des *ChWS5* et *ChWS6*. Deux molécules de la forme *Pass :ChWS7 :⟨ Result :ChWSi :Ri, Discriminator :Yes ⟩* sont alors produites (voir les lignes 11.01 à 11.05). Dans *ChWS5* et *ChWS6*, la règle *passInfo* (C) propage la molécule résultat de leur invocation respective vers *ChWS7* (voir les lignes 11.01 à 11.11). Une fois reçues, la règle *startExecDiscriminator* (I) est consommé par une réaction avec la première molécule *Discriminator :Yes* reçue soit de *ChWS5*, soit de *ChWS6*, permettant ainsi le *discriminateur*. De même, il déclenche une réaction avec la molécule *invokeServ* qui appelle le WS encapsulé par *ChWS7* produisant le résultat final du *workflow*.

5. Travaux connexes

Des travaux récents abordent le sujet de la décentralisation de l'exécution des *workflows*. Une idée récurrente est de remplacer le moteur centralisé BPEL par un ensemble de nœuds faiblement couplés coopérant. Une solution possible est d'utiliser un espace partagé en lecture et écriture contenant les informations nécessaires à la coordination des services [8, 9, 10]. Si ces approches sont proches de la notre, elles sont toujours basés sur des langages offrant un faible niveau d'abstraction. En particulier,

9.01 ChWS1 :⟨Dest :"ChWS2", Dest :"ChWS3", **Result :ChWS1 :R1**,
9.02 passInfo, addResult⟩,
9.03 ChWS2 :⟨Dest :"ChWS4", passInfo, addResult, invokeServ⟩,
9.04 ChWS3 :⟨Dest :"ChWS4", passInfo, addResult, invokeServ⟩,
9.05 ChWS4 :⟨In :2, passInfoCond, synchronize, syncFinished,
9.06 Start_invocation :⟨ ω ⟩, addResult, invokeServ
9.07 (replace-one condition_param by Condition_Pass :1,
9.08 Dest :"ChWS5", Dest :"ChWS6" if (condition))⟩,
9.09 ...

↓

9.10 ChWS1 :⟨**Pass :ChWS2 :**⟨**Result :ChWS1 :R1**⟩,
9.11 **Pass :ChWS2 :**⟨**Result :ChWS1 :R1**⟩, **passInfo**⟩,
9.12 ChWS2 :⟨Dest :"ChWS4", passInfo, addResult, invokeServ⟩,
9.13 ChWS3 :⟨Dest :"ChWS4", passInfo, addResult, invokeServ⟩,
9.14 ChWS4 :⟨In :2, passInfoCond, synchronize, syncFinished,
9.15 Start_invocation :⟨ ω ⟩, invokeServ, addResult,
9.16 (replace-one condition_param by Condition_Pass :1,
9.17 Dest :"ChWS5", Dest :"ChWS6" if (condition))⟩,
9.18 ...

9.19 ChWS1 :⟨...⟩,
9.20 ChWS2 :⟨Dest :"ChWS4", **Result :ChWS2 :R2**, passInfo, addResult⟩,
9.21 ChWS3 :⟨Dest :"ChWS4", **Result :ChWS3 :R3**, passInfo, addResult⟩,
9.22 ChWS4 :⟨In :2, passInfoCond, synchronize, syncFinished,
9.23 Start_invocation :⟨ ω ⟩, addResult, invokeServ
9.24 (replace-one condition_param by Condition_Pass :1,
9.25 Dest :"ChWS5", Dest :"ChWS6" if (condition))⟩,
9.26 ...

↓

9.27 ChWS1 :⟨...⟩,
9.28 ChWS2 :⟨**Pass :ChWS4 :**⟨**Result :ChWS2 :R2**⟩, **passInfo**⟩,
9.29 ChWS3 :⟨**Pass :ChWS4 :**⟨**Result :ChWS2 :R2**⟩, **passInfo**⟩,
9.30 ChWS4 :⟨In :2, passInfoCond, synchronize, syncFinished,
9.31 Start_invocation :⟨ ω ⟩, addResult, invokeServ
9.32 (replace-one condition_param by Condition_Pass :1,
9.33 Dest :"ChWS5", Dest :"ChWS6" if (condition))⟩,
9.34 ...

↓

9.35 ChWS1 :⟨...⟩, ChWS2 :⟨...⟩, ChWS3 :⟨...⟩,
9.36 ChWS4 :⟨**In :0**, passInfoCond, **synchronize**, **syncFinished**, addResult,
9.37 **Start_invocation :**⟨**Result :ChWS2 :R2, Result :ChWS3 :R2**⟩, Invoke :1
9.38 (replace-one condition_param by Condition_Pass :1,
9.39 Dest :"ChWS5", Dest :"ChWS6" if(condition)), invokeServ ⟩,
9.40 ...

FIGURE 10 – Exécution du workflow, étapes 1-5.

10.01 ChWS1 :⟨...⟩, ChWS2 :⟨...⟩, ChWS3 :⟨...⟩,
10.02 ChWS4 :⟨In :0, passInfoCond, synchronize, syncFinished, addResult
10.03 Start_invocation :⟨Result :ChWS2 :R2, Result :ChWS3 :R2⟩,
10.04 **Result :ChWS4 :R4**, **(replace-one condition_param by Condition_Pass :1,**
10.05 **Dest :"ChWS5", Dest :"ChWS6" if (condition))**⟩,
10.06 ...

↓

10.07 ChWS1 :⟨...⟩, ChWS2 :⟨...⟩, ChWS3 :⟨...⟩,
10.08 ChWS4 :⟨**Pass :ChWS5 :**⟨**Result :ChWS4 :R4**⟩,**passInfoCond**
10.09 **Pass :ChWS6 :**⟨**Result :ChWS4 :R4**⟩,**...**⟩,
10.10 ...

↓

10.11 ChWS4 :⟨...⟩,
10.12 ChWS5 :⟨**Dest :"ChWS7"**, **Result :ChWS4 :R4**, invokeServ
10.13 passInfo, **putDiscrInPass**⟩,
10.14 ChWS6 :⟨**Dest :"ChWS7"**, **Result :ChWS4 :R4**, invokeServ
10.15 passInfo, **putDiscrInPass**⟩,
10.16 ChWS7 :⟨invokeServ, startExecDiscriminator⟩

11.01 ChWS4 :⟨...⟩,
11.02 ChWS5 :⟨**Pass :ChWS7 :**⟨**Result :ChWS5 :R5,**
11.03 Discriminator :Yes⟩, **passInfo**⟩,
11.04 ChWS6 :⟨Pass :ChWS7 :⟨Result :ChWS6 :R6,
11.05 Discriminator :Yes⟩, passInfo⟩,
11.06 ChWS7 :⟨invokeServ, startExecDiscriminator⟩

↓

11.07 ChWS4 :⟨...⟩, ChWS5 :⟨...⟩,
11.08 ChWS6 :⟨Pass :ChWS7 :⟨Result :ChWS6 :R6,
11.09 Discriminator :Yes⟩,passInfo⟩,
11.10 ChWS7 :⟨invokeServ, **Result :ChWS5 :R5**,
11.11 **Discriminator :Yes**,**startExecDiscriminator**⟩

↓

11.12 ChWS4 :⟨...⟩, ChWS5 :⟨...⟩,
11.13 ChWS6 :⟨Pass :ChWS7 :⟨Result :ChWS6 :R6,
11.14 Discriminator :Yes⟩,passInfo⟩,
11.15 ChWS7 :⟨**Result :ChWS7 :R7**, ...⟩

FIGURE 11 – Exécution du workflow, étapes 6-11.

BPEL [11] ne peut exprimer de comportements dynamiques ou des exécutions décentralisées. Enfin, le partitionnement d'un processus BPEL est une tâche complexe à réaliser avant l'exécution.
Plusieurs langages ont été proposés pour exprimer des comportements décentralisés [12], qui sont encore traduit vers BPEL pour l'exécution, perdant ainsi le profit de tels langages. Du côté des *workflows* scientifiques, Scufl [13], DAX [14] ou GWorkflowDL [15] sont des langages permettant d'exprimer la coordination décentralisée. Ces langages sont plus simples et intuitifs, par l'expression du *workflow* par les dépendences de données et l'expression implicite du parallèlisme. Ils offrent cependant une expressivité limitée ne permettant pas de coder des *patterns* plus complexes tels que ceux que nous avons traité dans cet article, à l'exception de GWorkflowDL qui peut exprimer à la fois des flux de données et de contrôle. Cependant, les plates-formes d'exécution proposées de ces langages comme Taverna [16] ou Pegasus [14] sont toujours centralisées.

6. Conclusion

Le *design* et l'implémentation des systèmes de gestion de *workflow* est un sujet de recherche imprtant. Les solutions proposées sont pour la plupart fondée sur un modèle de coordination centralisé. De même, les langages de *workflow* de ces systèmes tels que BPEL ou XPDL sont intrinsèquement statiques et centralisés. D'autres langages développés dans un contexte de calcul scientifique, tels que Scufl ou DAX, même lorsqu'ils sont exécutés dans des environnements distribués, présentent une expressivité limitée, les empêchant de décrire des schémas de *workflow* complexes. Nous avons ici proposé un modèle de coordination de services décentralisée capable de coder des *patterns* complexes. Pour cela, nous nous sommes appuyés sur une métaphore chimique permettant d'exprimer des comportements autonomes et dynamiques par *composition moléculaire distribuée*. Par cette composition, une large variété de schémas de workflows, ainsi que leur exécution décentralisée peuvent être exprimées.

Bibliographie

1. M. Rosen, B. Lublinsky, K. T. Smith, and M. J. Balcer, Applied SOA : Service-Oriented Architecture and Design Strategies. Wiley, Jun. 2008.
2. G. Alonso, C. Mohan, D. Agrawal, and A. E. Abbadi. – Functionality and limitations of current workflow management systems. – IEEE Expert, vol. 12, 1997.
3. M. Viroli and F. Zambonelli. –A biochemical approach to adaptive service ecosystems.– Information Sciences, pp. 1 17, 2009.
4. J.-P. Banâtre, T. Priol, and Y. Radenac. –Chemical Programming of Future Service-oriented Architectures.– Journal of Software, vol. 4, no. 7, pp. 738 746, 2009.
5. J.-P. Banâtre, P. Fradet, and Y. Radenac. –Generalised multisets for chemical programming.– Mathematical Structures in Computer Science, vol. 16, no. 4, pp. 557 580, 2006.
6. H. Fernández, T. Priol, and C. Tedeschi. –Decentralized Approach for Execution of Composite Web Services using the Chemical Paradigm.– in ICWS 2010. Miami, USA : IEEE, 2010.
7. J.-P. Banâtre and D. Le Métayer. –The gamma model and its discipline of programming.– Sci. Comput. Program., vol. 15, no. 1, pp. 55 77, 1990.
8. P. A. Buhler and J. M. Vidal. –Enacting BPEL4WS specified workflows with multiagent systems.– In Proceedings of the Workshop on Web Services and Agent-Based Engineering, 2004.
9. C. Kunze, S. Zaplata, and W. Lamersdorf. –Mobile process description and execution.– in Distributed Applications and Interoperable Systems, 2006, pp. 32-47.
10. M. Sonntag, K. Gorlach, D. Karastoyanova, F. Leymann, and M. Reiter. –Process space-based scientific workflow enactment.– International Journal of Business Process Integration and Management, vol. 5, no. 1, 2010.
11. OASIS Consortium. –Web services business process execution language.– (WS-BPEL), Version 2.0, 2007.
12. I. J. Taylor, E. Deelman, D. B. Gannon, M. Shields, and A. Slominski. –Adapting BPEL to scientific workflows.– in Workflows for e-Science. Springer London, 2007, pp. 208 226.
13. W. Tan, P. Missier, I. Foster, R. Madduri, D. D. Roure, and C. Goble. –A comparison of using taverna and BPEL in building scientific workflows : the case of caGrid.– Concurr. Comput. : Pract. Exper., vol. 22, no. 9, 2010.
14. E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. –Pegasus : A framework for mapping complex scientific workflows onto distributed systems.– Sci. Program., vol. 13, no. 3, pp. 219 237, 2005.
15. M. Alt, S. Gorlatch, A. Hoheisel, and H. werner Pohl. –A grid workflow language using high-level petri nets.– vol. 3911, pp.715 722, 2005.
16. W. Tan, P. Missier, R. Madduri, and I. Foster. –Building scientific workflow with taverna and BPEL : a comparative study in caGrid.– in ICSOC 2008 International Workshops, Sydney, Australia. Springer-Verlag, 2009.