Pews Platform

Un environnement de composition de services web.

Cheikh Ba

Département d'Informtique
Université François Rabelais Tours
FRANCE
cheikh.ba.sn@gmail.com

RÉSUMÉ. PEWS (*Predicate Path Expression for web services*) est un langage permettant de définir l'interface comportementale de services web aussi bien simples que composés. Un service web simple est construit à partir des opérations définies dans un fichier de description *WSDL*. Un service composé est construit à partir d'autres services déjà définis. Les opérateurs de PEWS permettent en effet de définir l'ordre dans lequel les services et les opérations doivent être invoqués. Cet article présente l'environnement d'édition de programmes PEWS. Cet environnement se présente sous la forme d'une interface web permettant d'éditer, de tester et d'exécuter des spécifications de composition. L'interface web (*front-end*) interagit avec une application côté serveur (*back-end*) pour l'analyse des spécifications. Étant donné une spécification satisfiable, notre plateforme peut générer un squelette de programme java directement exécutable.

ABSTRACT. PEWS (*Predicate Path Expression for web services*) is a programming language for the definition of both simple and composite web service (behavioural) interfaces. Simple web services can be built from scratch, by the combination of (*WSDL*) operations. Composite web services are constructed from the combination of existing web services. PEWS operators help to define the order in which web services and operations will be performed. This paper presents the programming environment for PEWS. This environment is a web interface that helps users to easily develop, test and execute their composition specifications. The web interface (front-end) intracts with a server side application (back-end), for the specification analysis. Given a sound composition, our plateform can generate a java squeletton for execution purposes.

MOTS-CLÉS : Services web, composition, graphes de dépendance

KEYWORDS : Web services, composition, dependence graphs

1. Introduction

Un service Web est un composant logiciel autonome qui offre des services à travers une interface standardisée. La particularité d'une plateforme de services Web réside dans le fait qu'elle utilise la technologie Internet comme infrastructure pour la communication entre les composants logiciels (ou services Web), et ceci en mettant en place un cadre de travail basé sur un ensemble de standards. Le processus de standardisation est actuellement composé de trois principales couches : un protocole de communication SOAP (*Simple Object Acces Protocol*) permettant de structurer les messages échangés entre les composants logiciels, une spécification de description des interfaces des services WSDL (*Web Service Description Language*) et enfin une spécification de publication et de localisation de services UDDI (*Universal Description Discovery and Integration*).

Cependant, WSDL considère seulement les aspects statiques d'une interface (c'est à dire, la liste des opérations avec les messages d'entrée et de sortie) et non l'aspect dynamique représentant le comportement du service (l'ordre dans lequel les opérations peuvent être invoquées). Ce qui est une limitation pour les composants logiciels nécessitant l'exécution d'une interaction plus complexe qui obéit à un protocole spécifique de communication en vue de parvenir à la fonctionnalité attendue. C'est dans ce cadre que nous avons proposé un nouveau langage PEWS [4, 1], comme complément de WSDL, permettant de décrire l'interface dynamique des services web. Pour pouvoir analyser nos spécifications [2, 3], nous avons utilisé la théorie des traces de Mazurkievicz [6, 7].

Ce papier présente l'implémentation de l'état actuel de nos travaux. Cet implémentation se présente sous la forme d'une interface web permettant d'éditer, de tester et d'exécuter des programmes. Après une brève présentation de Pews, et de son interprétation par la théorie des traces, nous exposerons les différentes fonctionnalités de notre plateforme.

2. Pews

Différents mécanismes pour la synchronisation de processus ont été proposés comme langage pour la spécification des interfaces de services. Dans ce cadre, nous avons utilisé une notation basée sur les *predicate path expressions* [8] pour spécifier le comportement des services web en PEWS. Les *predicate path expressions* découlent de l'enrichissement des *path expressions* par des *prédicats*. Un prédicat est une fonction propositionnelle, ou une expression qui contient une ou plusieurs variables, et qui est vraie ou fausse selon la valeur qu'on attribue à celles-ci, ou selon les quantificateurs qui les lient.

Les *Path Expressions* sont des concepts de langage de programmation utilisés pour restreindre les séquences d'appel permises d'opérations (méthodes) d'un objet. Elles ont été créées pour spécifier la synchronisation de processus. Ces expressions, similaires à des expressions régulières, permettent une description de l'ordre dans lequel les opérations sont exécutées. Par exemple, étant donné les opérations a, b, et c, l'expression $a^* \cdot (b \parallel c)$ signifie que l'exécution parallèle des opérations b et c doit être précédée par zéro ou plusieurs exécutions de a.

Les *Predicate Path Expressions* étendent les expressions de chemins basiques en leur ajoutant des prédicats. Donc les prédicats sont une puissante extension des *path expressions*, et augmente leur expressivité, ce qui permet un contrôle plus fin des expressions de chemin à prédicats. Par exemple l'expression $a^* \cdot ([P]b + [not\ P]c)$, où P est un prédicat, indique que b ou c sera exécuté (après a^*) selon la valeur de vérité de P.

Nous avons proposé PEWS (*Path Expressions for Web Services*) [1, 4] en 2005 pour permettre de spécifier l'ordre dans lequel les opérations d'un service donné peuvent être exécutées. PEWS étend les *Predicate Path Expressions* en prenant en compte la notion de temps. Nous rejoignons l'idée selon laquelle il est important de considérer le temps lorsqu'on traite les services web. Cette notion de temps est d'autant plus importante que quand l'on veut comprendre les dynamiques des transactions et des compositions.

Pour donner un aperçu (non formel) de notre grammaire, nous considérons l'exemple d'un service d'entrepôt souhaitant que le paiement d'un produit soit fait dans les quarante-huit heures qui suivent l'envoi de la facture. Et si le delai n'est pas respecté, tout le processus d'achat est annulé. Un tel comportement de l'interface peut être spécifié par le programme PEWS suivant :

```
ns serviceNs = http://localhost:8080/pews/wsdl/Warehouse.wsdl
alias order = orderOper (input messOrder) in serviceNs
alias bill = billOper (output messBill) in serviceNs
alias payment = paymentOper (input messPayment) in serviceNs
alias receipt = receiptOper (output messReceipt) in serviceNs
alias abort = abortOper () in serviceNs
def tPay = now() - term(bill).time
order.bill.([tPay <= 48]payment.receipt + [tPay > 48]abort)
```

La valeur de `tPay` est calculée en évaluant l'expression `"now()-term(bill).time"`. Cette évaluation implique l'appel de la fonction `now()` qui renvoie la valeur courante du temps. Les prédicats qui apparaissent dans l'expression représentent des contrôles, donnant ainsi à l'expression une sémantique de choix conditionnel. Ces contrôles (prédicats) sont évalués jusqu'à ce que l'un d'eux soit vrai, c'est donc le principe de l'attente active. Ainsi, c'est la suite d'appel correspondant au prédicat évalué à vrai qui sera considérée. Notons que la valeur de la variable `tPay` change à chaque évaluation puisqu'elle dépend de la valeur courante du temps.

3. La théorie des traces

Dans cette section, nous donnons des notions très superficielles de la théorie des traces [6, 7], que nous avons utilisée pour l'analyse de nos spécifications en Pews [2, 3].

Mazurkiewicz a proposé en 1977 un modéle de parallélisme consistant à considérer des événements comme des symboles [5]. Ce modèle des traces est depuis plus de 20 ans un cadre formel fondamental de la recherche sur la modélisation et la vérification de systèmes distribués. Deux événements pouvant s'effectuer concurremment définissent un couple de symboles qui peuvent commuter. En effet, l'indépendance de certains événements dans un système concurrent permet d'avoir différentes suites d'événements pour représenter une seule et même exécution. Ainsi, un modèle fondé sur la représentation intégrale de ces suites d'événements équivalents n'est pas efficace. Le modèle des traces représente donc ces suites par des classes d'équivalence. Chaque classe représentera à elle seule un ensemble de comportements équivalents du système. La théorie des traces offre deux manières de représenter ces classes d'équivalence, et ces manières diffèrent par leurs commodités selon l'utilisation qu'on veut faire de ces classes. Une première représentation est celle qui consiste à utiliser une chaîne de symboles (ou *string*) pour représenter une classe d'équivalence. Cette représentation semble plus intuitive pour représenter un enchaînement d'événements, et de surcroît, dans la définition de la sémantique de PEWS.

L'autre représentation est celle qui consiste en l'utilisation d'un graphe de dépendance, dont quelques caractéristiques seront décrites dans la sous section suivante. Cette dernière représentation rend algorithmiquement plus efficace l'implémentation des opérateurs en Pews. En réalité, ces deux points de vue sont complémentaires, et permettent de se faire une idée à la fois plus complète et plus intéressante de la notion de trace : certaines propriétés sont plus faciles à établir lorsque l'on considère les traces comme des classes d'équivalence de mots, alors que d'autres sembleront plus naturelles lorsque l'on considère les traces comme des graphes.

Dans notre travail, les événements représentent les messages des services web. Chaque message de l'alphabet est lié aux décorations ?, ! et ♮ pour indiquer qu'il peut être, respectivement, un message en entrée, en sortie ou interne d'un service web. Un message en entrée est un message que le service attend, et il correspond (dans le fichier WSDL) au paramètre d'entrée d'une opération (ou méthode) du service. Un message en sortie est un message que le service envoie, et il correspond à l'élément de retour (paramètre de sortie) d'une opération du service. Un message interne correspond au résultat du *matching* (ou appariement) d'un message et de son complémentaire. Deux messages a et b sont dits *complémentaires* (noté $a = \overline{b}$) si et seulement si $a = m!$ et $b = m?$ ou vice versa

3.1. Les graphes de dépendance

Au lieu de voir une trace comme un ensemble de mots (ou chaînes de symboles), nous pouvons la voir comme un unique objet : un *graphe de dépendance*. Les graphes de dépendance sont des représentations graphiques de chaînes de symboles qui rendent explicite l'ordre de l'occurrence de ces symboles dans les traces. Il s'agit d'un graphe fini, dirigé, acyclique et dont les nœuds sont étiquetés par des labels (ou symboles) dans les mots. Dans ce graphe, deux nœuds sont liés par un arc si et seulement si ils sont des nœuds distincts et étiquetés par des symboles dépendants[1].

Nous transformons tout programme Pews en un *TSDG* (*Trace System Dependence Graphs*), qui est un ensemble de graphe de dépendances, dont chaque élément représente un comportement possible du système. L'absence de graphe signifie que la composition décrite par le programme Pews n'est pas satisfiable. La transformation d'une expression Pews en en un *TSDG* se fait de manière inductive sur l'expression [3, 5].

4. La plateforme Pews

Dans cette section, nous présentons *Pews Platform*, la plateforme d'édition, d'analyse et d'exécution de spécifications Pews. Cette interface offre plusieurs fonctionnalités qui seront présentées dans les sous sections suivantes.

4.1. Le générateur de codes Pews

Pour faciliter une saisie rapide et correcte d'un programme Pews, notre interface offre une fonctionnalité permettant à l'utilisateur d'avoir la portion du programme correspondant à un service web, défini par sa description WSDL. L'utilisateur doit renseigner une zone de saisie en y mettant l'URL qui pointe vers la ressource WSDL. En cliquant sur le bouton `"Load wsdl data"`, le contenu de la zone de texte contenant le programme

1. Une dépendance entre deux messages d'un service web peut être créé si l'on impose un ordre dans lequel les deux événement correspondants doivent se produire

est enrichi par une définition d'un espace de noms (`ns ...`) correspondant au fichier WSDL, et par la liste des définitions (`alias ...`) des opérations offertes par le service. La figure 1 montre un exemple de code généré à partir du service web décrit par le fichier WSDL `Catalogue.wsdl`.

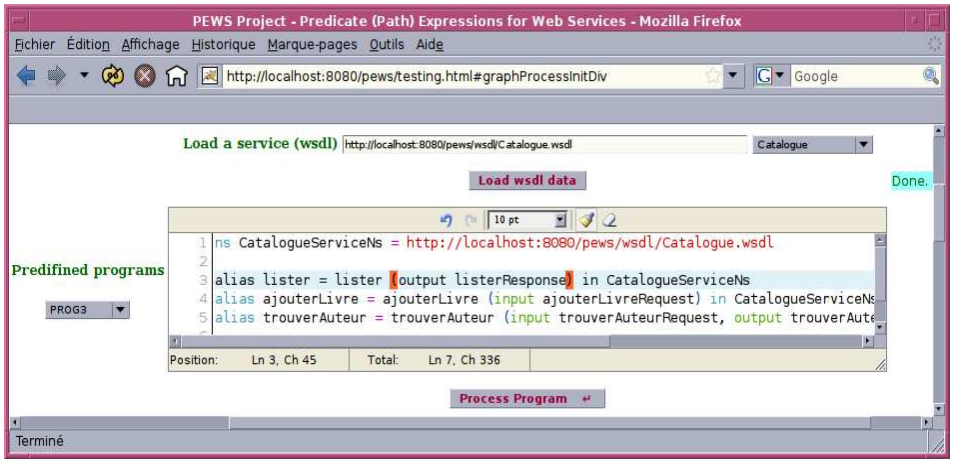

Figure 1. *Génération de codes Pews.*

L'utilisateur peut utiliser cette fonctionnalité pour tous les services qui seront utilisés dans la composition. Il lui faudra par la suite compléter le code généré par d'éventuelles définitions de variables, et par la définition obligatoire de l'expression Pews représentant la composition. Il est à noter aussi qu'il y a trois programmes Pews complets que l'utilisateur peut charger en guise de test de la plateforme. On peut les choisir en utilisant la liste déroulante `"Predifined programs"` à gauche de l'éditeur.

4.2. L'éditeur de programmes

L'éditeur est la principale interface avec l'utilisateur, dans laquelle la spécification de la composition est faite. Il se présente sous la forme d'une zone de texte HTML, enrichie par la bibliothèque javascript *EditArea*, configurée pour la prise en compte de notre langage. Le premier avantage de cet éditeur est donc l'apport visuel concernant la coloration syntaxique (c'est à dire, une surbrillance des mots réservés, des chaînes de caractères, des commentaires, etc.), la signalisation des délimitations de blocs de code, et la possibilité de redimensionner la zone d'édition. L'autre avantage, comme nous pouvons le voir sur la figure 1, est la présence d'un menu permettant de défaire ou refaire du texte, de modifier la taille du texte dans la zone et, dans une moindre mesure, d'enlever ou de remettre la coloration syntaxique. Ce sont donc des fonctionnalités qui existent dans bon nombre d'environnements de développement intégré, mais que nous avons su reproduire dans le contexte d'une interface web.

4.3. Le compilateur

Le compilateur permet de faire les analyses lexicales et syntaxiques de programmes PEWS. Il a été créé en utilisant JavaCC[2] (*Java Compiler Compiler* de Sun Microsystems), qui est une API libre permettant de générer un parser en langage java. Le lancement de la compilation se fait en cliquant sur le bouton `"Process Program"` (voir figure 1).

Les erreurs de compilation peuvent être de plusieurs sortes. Une erreur lexicale survient quand le programme contient un mot qui n'est pas reconnu par la grammaire Pews. Une erreur syntaxique est obtenue lorsqu'une expression n'a pas une syntaxe valide par rapport à la grammaire Pews. D'autres erreurs peuvent survenir quand un espace de noms (`ns`) ne pointe pas sur un fichier WSDL, quand une déclaration d'un `alias` fait référence à un espace de noms non déclaré, quand le nom d'une opération du programme ne correspond pas à un vrai nom d'opération dans le fichier WSDL en question, ou quand les types d'opération dans le programme ne correspondent pas aux vrais types d'opération dans les fichiers WSDL. Pour chaque erreur dans le processus de compilation, un message est fourni à l'utilisateur pour lui indiquer l'origine (numéros de ligne et colonne) et les raisons de l'erreur, afin de lui faciliter la correction.

S'il n'y a pas d'erreur de compilation, le programme Pews est alors correct. Le programme est alors transformé en un *arbre syntaxique*. Cet arbre syntaxique est une sorte de passerelle entre la description PEWS d'un service composé et nos algorithmes. Cet arbre fait ressortir de manière claire les règles d'évaluation d'une expression PEWS en termes de priorités des opérateurs. Un arbre syntaxique est un arbre binaire, dont les nœuds sont des opérateurs Pews ou des prédicats, et les feuilles sont des opérations de services Web. Le code permettant de générer l'arbre syntaxique est fait de manière générique, afin qu'il puisse s'adapter à des changements sur le nombre d'opérateurs ou sur leurs priorités.

L'arbre syntaxique est la principale structure de données (pour nos algorithmes) qui représente fidèlement toutes les informations de la spécification à analyser. Nous l'utilisons pour générer les graphes de dépendance pour la partie concernant l'analyse statique de nos spécification. L'existence d'un graphe de dépendance signifie qu'il existe au moins une possibilité pour les services Web de coopérer conformément à la spécification, tout en respectant leurs comportements. Chaque graphe de dépendance décrit donc l'évolution du processus modélisé, c'est à dire l'ordre dans lequel les messages doivent être traités pendant l'exécution. S'il n'y a pas d'erreur pendant la compilation, l'interface affiche tous les graphes de dépendance correspondant à la spéficication, comme nous pouvons le voir avec l'exemple sur la figure 2.

4.4. Le moteur d'exécution

Il est possible pour l'utilisateur de choisir l'un des comportements possibles de sa spécification (comportement représenté par un graphe de dépendance) pour lancer son exécution. La sélection se fait en cliquant sur le graphe choisi. L'exécution consiste à faire un parcours du graphe par ses nœuds minimaux[3]. Pour chaque nœud minimal rencontré, et selon le type du message correspondant, nous faisons une action appropriée. Par exemple, si nous tombons sur un message en entrée, nous présentons un formulaire à l'utilisateur pour qu'il saisisse des valeurs. Si nous avons un message en sortie, nous l'affichons. Et si

2. http ://javacc.dev.java.net/
3. Nous rappelons qu'un nœud minimal d'un graphe est un nœud qui n'a pas d'arc entrant.

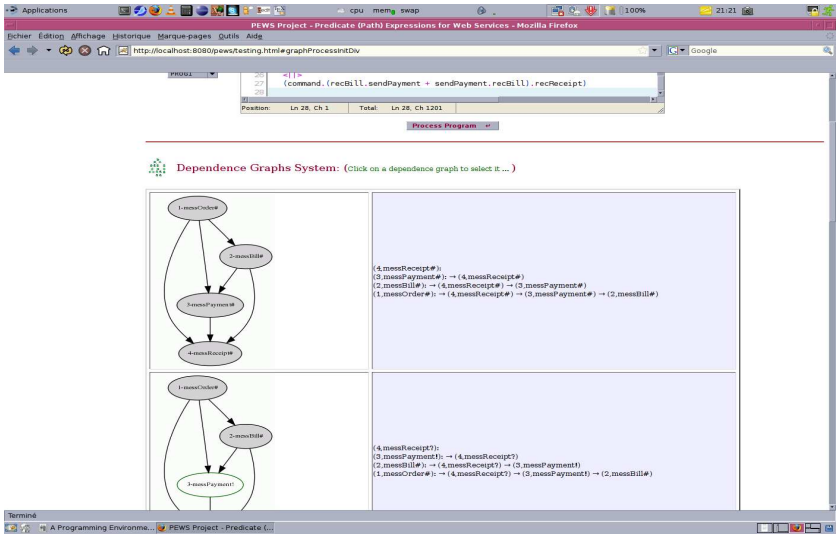

Figure 2. *Affichage de graphes correspondant à un programme Pews correct.*

nous avons un message interne, nous faisons, à l'insu du monde extérieur, l'appariement (*matching*) entre un message en sortie et le message en entrée correspondant. Ce processus est poursuivi jusqu'à ce qu'il ne reste plus de nœud dans le graphe. Notons que nous avons un système de *buffers* qui nous permet de gérer les données fournies par l'utilisateur, et celles reçues lors des appels de méthodes des services, pour une utilisation utltérieure.

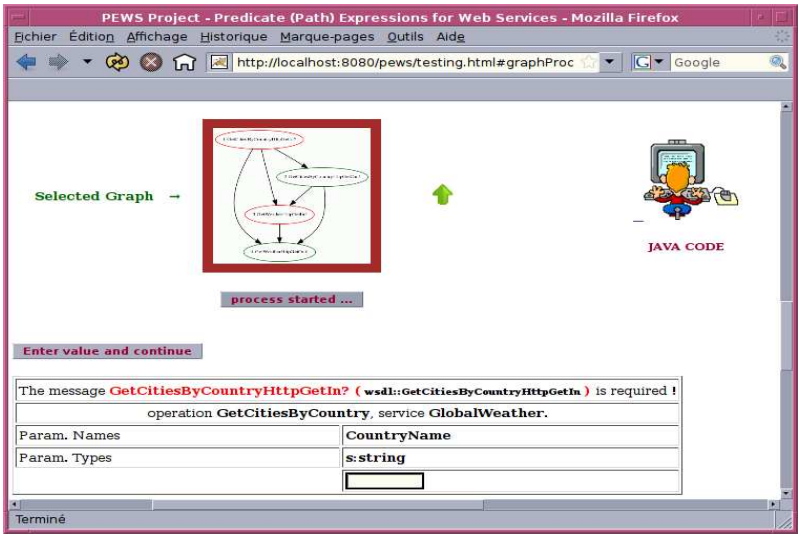

Figure 3. *Une étape d'exécution d'un graphe.*

La figure 3 montre un exemple d'exécution d'un graphe dont le premier nœud minimal est un message en entrée, correspondant au nom d'un pays dont on voudrait connaître les noms des villes ayant des informations météorologiques.

4.5. Le générateur de programme java

Un incovénient du moteur d'exécution présenté précédemment est le fait qu'il ne permet pas une gestion plus fine des données, c'est à dire une libérté d'action sur les messages échangés entre les services. Pour pallier cela, nous offrons la possibilité de générer, de manière automatique, un programme java, directement exécutable, et contenant l'ensemble des instructions pour la consommation du graphe. La génération se fait en cliquant tout simplement sur le lien `"JAVA CODE"` (voir figure 3). Une nouvelle page contenant le code java, ainsi que les instructions pour l'exécuter, est alors affichée. De cette manière, un développeur java pourra avoir la mainmise sur une gestion plus fine et personnalisée des données.

5. Conclusion

Nous avons présenté l'implémentation de l'état actuel de nos travaux [1, 2, 3, 4, 5] sur la composition de services web avec le langage Pews. Ce travail d'implémentation suivra naturellement l'évolution de notre principale proposition. En effet, plusieurs perspectives s'ouvrent à nous. Une première idée serait, par exemple, de travailler sur l'analyse dynamique de Pews, c'est à dire en prenant en compte les prédicats en *runtime*. On ne pouvait en effet pas les prendre en compte puisque les prédicats peuvent contenir des expressions ne pouvant être évaluées que pendant l'exécution. Nous prévoyons aussi de travailler sur les problématiques de recherche ou de remplacement d'un service dans une composition, sans pourtant compromettre la précédente coopération.

6. Bibliographie

[1] C. BA, M.A. CARRERO, M. HALFELD-FERRARI-ALVES, M. MUSICANTE, « PEWS : A New Language for Building Web Service Interfaces », *Journal of Universal Computer Science*, vol. 11, n° 7, pp. 1215-1233, 2005.

[2] C. BA, M. HALFELD-FERRARI-ALVES, M. MUSICANTE, « Composing Web Services with PEWS : a trace-theoretical approach », *The 4th IEEE European Conference on Web Services (ECOWS)*, pp. 65-74, 2006.

[3] C. BA, M. HALFELD-FERRARI, « Dependence Graphs for Verifications of Web Service Compositions with PEWS », *SAC'08 (The 23rd ACM Symposium on Applied Computing)*, pp. 2387-2391, March 2008, Fortaleza, Ceará, Brazil.

[4] C. BA, M. HALFELD-FERRARI-ALVES, M. MUSICANTE, « Building Web Service Interfaces using Predicate Path Expressions », *SBLP 2005 - 9th Brazilian Symposium on Programming*, BRAZIL, 2005.

[5] C. BA, « Composition de services Web avec PEWS : approche par la théorie des traces », *Thèse en Informatique, Université François Rabelais Tours*, novembre 2008.

[6] A. MAZURKIEWICZ, « Trace theory », *LNCS. Springer-Verlag*, n° 255, 1987.

[7] A. MAZURKIEWICZ, « Introduction to trace theory », *In Volker Diekert and Grzegorz Rozenberg, editors, The Book of Traces, World Scientific*, 1995.

[8] S. ANDLER, « Predicate path expressions. », *In Sixth Annual ACM Symposium on Principles of Programming Languages, 6th POPL'79*, 1979.