Un modèle de programmation de services pour le web sémantique

A service programming model for the semantic web

N. Sabouret[1] J. Pierson[1]

[1] LIP6
[2] LIMSI-CNRS

LIP6, 8, rue du Capitaine Scott, 75015 Paris
LIMSI, BP 133, 91403 Orsay Cedex

Résumé

Dans cet article, nous proposons un modèle de programmation de services web actifs en XML qui permet d'intégrer dans une même représentation la description de la partie structurelle d'un service (c'est-à-dire l'information manipulée) et celle de son fonctionnement (c'est-à-dire les actions de manipulation). Nous présentons un modèle d'activité et un langage de programmation qui permettent d'avoir accès en cours d'exécution *à la description du fonctionnement du service, par exemple pour répondre à des requêtes ce fonctionnement. Nous montrons comment ce modèle permet à la fois d'intégrer les standards émergeants de description de service (WSDL, DAML-S,* etc*) et de les étendre pour prendre en compte d'une part l'aspect proactifs des composants actifs dans le web sémantique et d'autre part l'interaction avec l'humain.*

Mots Clef

Web sémantique et web services, raisonnement sur les actions, composants actifs, interaction homme-machine.

Abstract

In this paper, we propose a programming model for web services and, beyond, active components, based on XML. This allows us to integrate in a single représentation both the structural data description and the action description. We present an execution model and a programming language that gives access at runtime *to the service's actions description, so as to be able to answer formal requests about actions, for instance. We show how this model can integrate currently emergent standards in web services description (WSDL, DAML-S,* etc*) and take into account the proactive nature of active components in the semantic web, especially for human-computer interaction.*

Keywords

Semantic Web Services, Reasoning about Actions, Active Software Components, Human-Computer Interaction.

1 Contexte de l'étude

1.1 Les composants actifs dans le web sémantique

Les modèles actuels du web sémantique proposent de munir l'information sur le web d'une sémantique afin de la rendre manipulable par des composants logiciels actifs (services, agent, *etc*). Ces composants sont caractérisés par un *fonctionnement* [5] et, contrairement au « script CGI » qui ne fait que traiter, manipuler ou produire l'information contenue sur le web, ce fonctionnement fait *partie intégrante* de l'information. Les utilisateurs de tels composants actifs (qu'il s'agisse d'humains ou d'agents) doivent alors pouvoir les interroger non seulement à propos des données qu'ils manipulent, mais aussi sur à propos des actions qu'ils effectuent.

Pour ce faire, le web sémantique doit permettre de représenter et de manipuler des connaissances non seulement structurelles, mais aussi de nature *fonctionnelle*, c'est-à-dire de représenter le fonctionnement des composants actifs qui y agissent. Les standards émergeants de description de service web comme WSDL [2] et DAML-S [3] proposent de définir un ensemble de protocoles pour déclarer des services web (des ressources web offrant tels services et vérifiant telles propriétés du point de vue de l'exécution) et permettre aux agents de trouver, invoquer, surveiller ou composer en services plus complexes ces ressources. Cependant, cette approche soulève plusieurs problèmes :

– La description du service est séparée de l'information manipulée. Au contraire, dans un composant actif, la

structure, les données manipulées et le fonctionnement doivent être intégrés au sein d'une même représentation pour que le composant puisse interagir avec l'utilisateur (agent ou humain) aussi bien à propos de ses données [1] que de sont fonctionnement [7]. Il faut donc pouvoir *programmer* directement en XML les composants actifs, afin qu'ils soient capables de raisonner sur *leur propre code*.

– Les actions de manipulation des données par le service sont vues comme des processus *atomiques* (approche « boîte noire »). Par conséquent, il n'est pas possible de poser des questions sur les opérations effectuées ou sur le fonctionnement du service en dehors de ce qui a été spécifié explicitement dans les fichiers DAML-S. Au contraire, notre objectif est de permettre aux composants actifs dans le web sémantique d'interagir à propos de ce qu'il font pour fournir des explications sur le propre fonctionnement.
– Les services sont uniquement *réactifs* : ils traitent de l'information à la manière des scripts CGI. Au contraire, les composants dans le web sémantique doivent être proactifs et dépasser la notion de service afin d'interagir d'eux-même avec l'utilisateur (par exemple pour la surveillance de système physiques via des interfaces sur internet).

1.2 Travaux antérieurs

Dans le cadre du projet *InterViews* [11], nous avons proposé un modèle de programmation et d'exécution de composants actifs en XML [10] qui vérifie les propriétés suivantes :

1. Il permet d'accéder *en cours d'exécution* à la description des actions des composants, afin de produire des explications sur le fonctionnement en réponse à des questions de l'utilisateur et en s'appuyant sur la sémantique opérationnelle du langage de description.
2. Les descriptions du fonctionnement et de la partie structurelle sont *intégrées* au sein d'une même représentation.

Dans ce modèle, baptisé VDL pour ***V**iew **D**esign **L**anguage*, la description d'un composant est un arbre (un document XML. L'exécution d'un composant consiste en la réécriture de l'arbre à chaque *cycle* en fonction de concepts spécifiques, comme dans logiques de réécriture [6]. Ce modèle est Turing-complet (c'est-à-dire qu'il est possible de modéliser n'importe quelle machine de Turing en VDL et donc, théoriquement, d'écrire n'importe quel programme en VDL) et linéaire en temps et en espace par rapport à la machine de Turing équivalente. Nous avons défini complètement la sémantique opérationnelle de ce langage dans [8] et nous l'avons implémenté en Java[1].

[1]Des démonstrations sont disponibles sur notre page web : `http://www.limsi.fr/Individu/nico/exemples`

Nous avons montré dans [9] que ce modèle permet de modéliser et d'implémenter des agents capables raisonner sur leur propre fonctionnement pour répondre *en cours d'exécution* aux questions que peut poser un utilisateur humain en situation de demande d'aide à propos de leurs actions et leur exécution. Par exemple : « *Qu'est-ce que tu fais ?* », « *Comment faire pour... ?* », « *Pourquoi tu as fait ça ?* », etc.

1.3 Plan de l'article

Dans cet article, nous montrons comment ce modèle de description de composants actifs en XML peut être utilisé pour la description d'un service web. Nous rappelons tout d'abord brièvement le modèle d'exécution VDL. Nous présentons ensuite l'architecture client-serveur HTTP dans laquelle sont implémentés les services. Nous montrons comment elle peut intégrer les modèles de description de services standards DAML-S et WSDL pour composer des services VDL avec d'autres services web.. Nous illustrons notre modèle à l'aide d'un exemple de service de location de DVD. En conclusion, nous discutons les possibilités nouvelles offertes par notre approche dans les domaine de l'utilisation (découverte et composition) des services et de l'interaction homme-machine.

2 Modèle d'exécution

2.1 Notion d'observateur

Le modèle d'exécution que nous proposons pour les composants actifs décrits en XML s'appuie sur la notion d'*observateur*, inspirée des SGML. Un observateur est un composant logiciel externe qui recherche dans un document XML des éléments spécifiques, conformes à un schéma de description ou à une DTD, et les interprète pour effectuer des actions particulières. La fonction d'interprétation d'un observateur définit les actions effectuées sur les éléments reconnus par l'observateur, c'est-à-dire la *sémantique* de ces éléments.
Du point de vue de notre étude, nous pouvons mettre en évidences deux types d'observateurs :

1. **Les observateurs statiques**, qui ne modifient pas le document de manière proactive. Par exemple, l'observateur structurel qui répond aux requêtes sémantiques portant sur les informations contenues dans le composant, l'observateur d'interface qui gère l'interface graphique utilisateur du composant (modifiée par effet de bord), l'observateur de base de données qui permet d'accéder aux données du site, *etc*.
2. **L'observateur dynamique** qui considère le document passé en paramètre comme la description du composant à l'instant t et qui construit sa description à l'instant suivant $t+1$ en fonction des éléments spécifiques décrivant le fonctionnement du composant actif.

L'exécution d'un composant actif dans ce schéma consiste donc en la réécriture du document XML. La description du composant en XML est sérialisée à chaque cycle d'exécution et chaque observateur (l'ensemble des observateurs statiques puis à nouveau l'observateur dynamique) interprète le nouveau document. De plus, les composants sont proactifs : ils peuvent s'exécuter en dehors de toute interaction avec l'utilisateur.
L'observateur dynamique que nous utilisons dans notre modèle est noté vdl, par référence aux travaux effectués dans le projet *InterViews*. Dans ce modèle d'exécution basé sur la réécriture d'arbre, le fonctionnement est décrit explicitement (à la fois dans l'abstraction XML et dans le document sérialisé à chaque cycle) et peut être manipulé pour répondre à des requêtes sur le fonctionnement, en utilisant la sémantique opérationnelle du langage définie par la fonction d'interprétation de l'observateur vdl. De nombreux exemples sont donnés dans [7].

2.2 Observateur VDL

Principe général. Le schéma de l'observateur d'exécution vdl, qui en définit la syntaxe, est disponible sur notre page web[2]. Nous rappelons ici les principaux éléments.
– Un composant actif est décrit dans notre modèle par un arbre XML de racine **view**.
– Les « valeurs » des variables (c'est-à-dire les données manipulées) sont, par définition, les données contenues dans les éléments, conformément aux recommandations du W3C.
Pour donner des valeurs dans la description du fonctionnement, nous les englobons dans des éléments **value**.
Les éléments $view$ et $value$ peuvent contenir des éléments quelconques, en particulier des éléments qui ne sont pas définis dans le schéma *vdl.xsd*. Par conséquent, notre modèle permet d'intégrer la description du fonctionnement au sein de n'importe quel document structuré pour en faire un composant actif.

2.3 Partie fonctionnelle

Références. Pour faire référence au sein d'une actions aux autres éléments du document pour les modifier ou pour en extraire leur valeur, nous proposons d'utiliser des références respectant la syntaxe *XPath*[3] définie dans la norme XML et englobés dans des éléments **path** ou **get**. L'élément $path$ permet de faire référence directement à un élément du composant pour le modifier. Il sera utilisé dans les *actions élémentaires*. Au contraire, l'élément get permet de faire référence à un élément pour extraire sa « valeur » dans les opérations arithmétiques.

Opérateurs arithmétiques. Les opérations arithmétiques de base sont modélisées à l'aide des éléments **plus**, **times**, **minus** et **inverse** pour les réels ; **equals** et **"greater than"** pour les relations ; **and**, **or**, **not**, **true** et **false** pour les booléens. Dans notre schéma *vdl.xsd*, l'ensemble de ces opérations arithmétiques, l'élément de calcul de valeur d'un terme get et l'élément de valeur terminale quelconque $value$ sont regroupées dans le groupe $operation$. Les éléments $value$ peuvent contenir soit des valeurs numériques, sérialisées par un contenu de type $string$, soit des éléments structurés définis dans un schéma ou une DTD spécifique au composant.
Nous avons défini dans [8] une interprétation canonique sur les termes. L'interprétation canonique d'une valeur ($value$) est l'interprétation canonique de son contenu (par exemple, 2 est interprété comme le nombre entier deux) ; l'interprétation canonique des éléments représentant des opérateurs arithmétiques est l'opération représentée ; l'interprétation canonique d'un élément get est le contenu (non interprété) de l'élément référencé dans le get. Ainsi, l'opération $1 + \frac{x}{2}$ est représentée dans notre langage par l'élément :

```
<plus>
  <value>1</value>
  <times>
    <get>//variable/x</get>
    <inverse><value>2</value></inverse>
  </times>
</plus>
```

Actions élémentaires. Il existe trois actions élémentaires de modification d'un document XML dans notre modèle [10] : **add** pour ajouter des fils à un élément, **put** pour remplacer tout le contenu (tous les fils) d'un élément, **del** pour supprimer un élément et tous ses fils. Les actions élémentaires doivent nécessairement contenir au moins un élément $path$. De plus, les éléments add et put doivent contenir au moins un élément du groupe $operation$ donnant les éléments à ajouter.

Préconditions. Chaque action élémentaire peut être munie de deux sortes de préconditions :
– Des gardes booléennes, définies dans un élément **guard** qui contient nécessairement des éléments du groupe $operation$ pouvant être évalués à $true$ ou $false$;
– Des événements externes, utilisés pour modéliser l'interaction avec un utilisateur humain ou avec d'autres composants, définis dans un élément **event**. Un événement externe est un élément XML qui peut être envoyé au composant au cours de son exécution. L'élément $event$ permet au programmeur de définir dans le composant quels événements doivent être traités et comment ils sont utilisés.
Les actions sont les éléments contenant au moins une action élémentaire, une précondition ou une sous-action. Ils sont matérialisés par l'élément **action**.
Une action n'est effectuée que si toutes ses préconditions sont vérifiées. L'ensemble des préconditions d'une

[2] http://www.limsi.fr/Individu/nico/xml/vdl.xsd
[3] http://www.w3.org/TR/xpath.html

```
<action> <name>action1</name>
  <guard> <get>//variable[@name="test"]</get> </guard>
  <action> <name>action2</name>
    <event><ok/></event>
    <del> <path>//un-terme</path> </del>
  </action>
  <put>
    <path>//variable[@name="compteur"]</path>
    <plus> <get>//variable[@name="compteur"]</get> <value>1</value> </plus>
  </put>
</action>
```

FIG. 1 – Préconditions d'une actions.
L'action $action1$ est effectuée dès lors que la variable « test » est évaluée à $true$. Elle incrémente la valeur de la variable « compteur » (action élémentaire put). De plus, si l'utilisateur a envoyé un événement externe « ok », la sous-action $action2$ supprime l'élément « un-terme » du document.

action est, par définition, l'ensemble des éléments $guard$ ou $event$ parmi ses fils. De plus, si cette action est sous-action d'une autre action, les préconditions de l'action parent sont aussi préconditions de l'action fille.
L'exemple donné figure 1 illustre ce principe.

2.4 Remarques

Les composants actifs décrits en VDL ont accès *en cours d'exécution* à une description de leur fonctionnement et de leur structure. Il leur est alors possible de raisonner dessus, par exemple pour répondre à des questions d'un utilisateur en situation de demande d'aide, à travers un observateur de traitement de requêtes sur les actions. Nous avons développé ces travaux dans [7].
Notre langage n'utilise pas les opérateurs usuels des langages de programmation, comme if, $while$, *etc*, issus du modèle *implémentatoire* classique « instruction-valeur ». Au contraire, comme les modèles de description et de composition de services web (WSFL[4], WSMF [4], DAML-S [3]...), il se situe *au niveau conceptuel*, dans lequel la sémantique des opérations effectuées est matérialisée par leurs *préconditions*. En particulier, notre modèle de description de composant permet de décrire toutes les constructions de process proposées par DAML-S pour la composition de service ($Sequence$, $Concurrent$, $Split$, $Split+Join$, $Unordered$, $Choice$, $If-Then-Else$, $Repeat-Until$ et $Repeat-While$). La différence fondamentale avec DAML-S est que la description du fonctionnement interne du composant n'est pas vue comme une « boite noire » : elle est intégrée avec la description du service (et inversement : la composition de service est elle-même décrite en VDL). Ainsi, le service peut répondre à des requêtes sur « comment » et « pourquoi » il invoque ou compose tel service à tel moment de l'exécution.
La seconde différence (corollaire de la première) est que les services décrits en VDL sont **actifs** et pas seulement réactifs, comme nous allons le voir dans la section suivante. Il se comportent alors plus comme des agents, capables de prendre des initiatives (par exemple pour avertir l'utilisateur d'un fait important ou pour rechercher des services ou découvrir dynamiquement les fonctionnalités opératoires d'un autre agent).

3 Exécution de services web

Dans cette section, nous présentons l'architecture générale d'exécution d'un service web en décrit en VDL ainsi que les observateurs qui entrent en jeu dans cette exécution.

3.1 Architecture générale

Dans notre modèle, le code du service web en VDL comprend à la fois les données manipulées par le composant, les actions de manipulation, la description de l'interface utilisateur et la description des capacités d'interaction avec d'autres services web. Il est défini dans une page XML conforme au schéma $vdl.xsd$.
Lorsque le service manipule de grosses quantité de données, elles-sont définies dans des pages XML spécifiques qui jouent le rôle de base de données. Les données contenues dans ces pages sont alors manipulées au sein du service à l'aide de requêtes XQuery[5].
Le modèle d'exécution repose sur trois observateurs :
– L'observateur d'exécution VDL qui réécrit le composant à chaque cycle d'exécution, conformément au modèle présenté dans la section précédente ;
– L'observateur d'interface HTML qui produit, à chaque cycle d'exécution, une page web en HTML correspondant à l'interface utilisateur du service ;
– L'observateur de déclaration du service web, qui génère et gère les fichiers de déclaration de service web en WSDL et DAML-S.

[4] http://www.ebpml.org/wsfl.htm

[5] http://www.w3.org/TR/xquery/

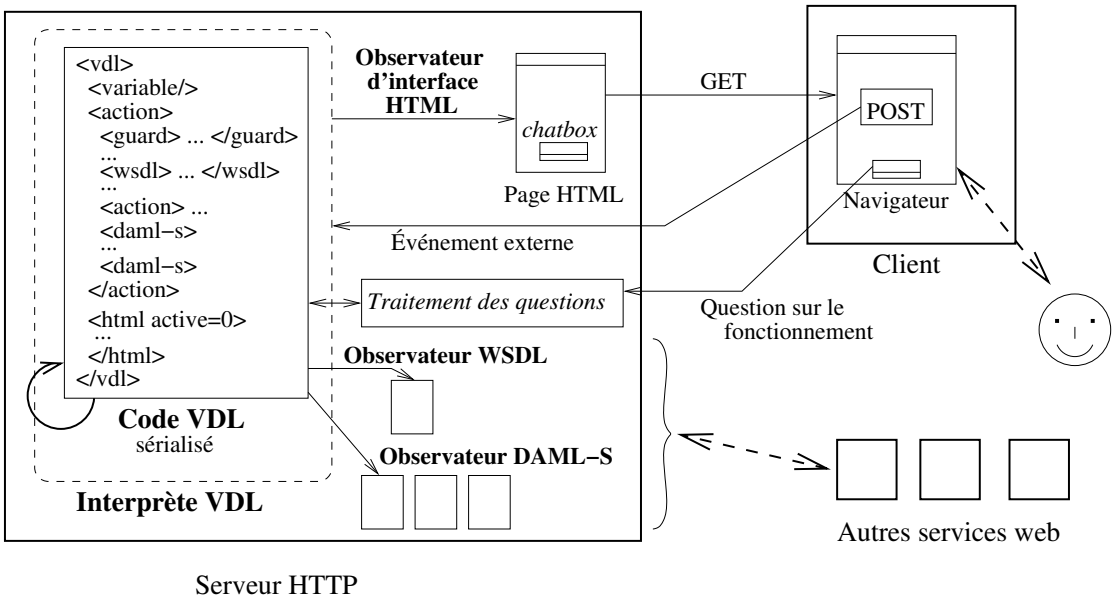

FIG. 2 – Architecture d'un service web en VDL.

La figure 2 illustre le fonctionnement global d'un service VDL.

3.2 Connexion au service

Chaque utilisateur (humain ou agent) du service interagit avec un instance différente du service. Lors de la connexion au service web, l'utilisateur appelle un script CGI qui :

1. Attribue un numéro unique d'identification à l'utilisateur.
2. Construit une copie de la description VDL initiale du code du service dans un fichier XML dont le nom est le numéro unique d'identification.
3. Lance l'exécution de l'interprète VDL sur cette instance. L'utilisateur (humain ou agent) va alors interagir avec cette instance du service, qui a été créée pour lui (et qui se modifie à chaque cycle d'exécution en fonction de l'interaction).
4. Pour les utilisateurs humains, le script retourne une page web (la page d'accueil du service) construite par appel de l'observateur d'interface HTML sur la description initiale du service.[6]

   Pour les utilisateurs agents (ou services), le numéro d'identification est donné en paramètre dans le message SOAP de retour.

[6]En fait, ce mécanisme doit être transparent pour l'utilisateur : le numéro d'identification est enregistré dans un cookie expirant à la fin de la session. Il est récupéré par le script CGI du service pour savoir quelle est la page affichée à l'utilisateur, qui a ainsi l'impression de rester sur la page principale du service.

Du point de vue de la déclaration et de l'invocation de ce service web par un agent, les fonctionnalités déclarées et enregistrées dans l'annuaire UDDI pour le service sont réduites à cette connexion initiale, ce qui correspond au « service profile » DAML-S donné figure 3.
Dès le début de l'exécution de l'instance du service[7], l'observateur de service web génère les descriptions DAML-S et WSDL spécifiques à l'instance, accessibles par le service utilisateur à l'aide du numéro d'identification. La description de ces fonctionnalités étant intégrée dans le code VDL du service actif, elles peuvent être modifiées au cours de l'exécution. L'objectif à terme est de permettre à un service actif de pouvoir s'adapter aux fonctionnalités opératoires de l'utilisateur au cours de l'interaction.

Déconnexion. Afin de prendre en compte la possibilité pour un utilisateur (humain ou service) de quitter le service sans l'avertir (par exemple par fermeture du navigateur), les instances de service VDL sont temporisées. Elles arrêtent de s'exécuter et disparaissent dès qu'elles ne reçoivent pas d'événement externe pendant une durée suffisamment importante.

3.3 Observateur d'exécution

L'observateur d'exécution est responsable de la réécriture du composant actif (*i.e.* l'instance du service) à chaque cycle d'exécution. Un service VDL n'est donc pas simplement réactif (comme un service web classique) : il est proactif. Il se comporte comme un agent qui s'exécute sur

[7]Dans la suite du document, nous parlerons de service VDL pour désigner une instance de service VDL en exécution.

```
<daml:Class rdf:ID="DVD_rent_service">
  <rdfs:subClassOf rdf:resource="...daml-s/profileHierachy#E_Commerce"/>
  <daml:comment>Class that represents DVD rent Services</daml:comment>
</daml:Class>

<daml:Class rdf:ID="DVD">
  <rdfs:subClassOf rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
  <rdfs:subClassOf rdf:resource="...daml-s/ProfileHierarchy.daml#Product"/>
</daml:Class>

<profileHierarchy:DVD_rent_service rdf:ID="DVD_rent">
  [...]
  <profile:output>
    <profile:ParameterDescription rdf:ID="No_identification">
      <profile:parameterName>No Identificationr</profile:parameterName>
      <profile:restrictedTo rdf:resource="http://www.DVDRent.com/process.daml#Id_number"/>
      <profile:refersTo rdf:resource="http://www.DVD_rent.com/process.daml#DVDRent_ID" />
    </profile:ParameterDescription>
  </profile:output>
<profileHierarchy:DVD_rent_service>
```

FIG. 3 – Architecture du « service profile » initial, utilisé lors de la connexion.

le serveur web (une instance par connection et par session) et interagit avec l'utilisateur ou d'autres agents ou services.

Événements externes. Les services VDL (et plus généralement les composants actifs VDL) interagissent avec l'extérieur à l'aide d'événements externes. Ces événements externes envoyés au service doivent être englobé dans un message SOAP de cette forme :

```
<?xml version="1.0" encoding="UTF-8"?>
  <SOAP:Envelope xmlns:SOAP=
        "http://schemas.xmlsoap.org/soap"/>
  ... en-têtes SOAP ...
  <SOAP:Body>
    ... L'événement externe (arbre XML) ...
  </SOAP:Body>
</SOAP:Envelope>
```

Ils sont produits soit par les autres agents ou services qui souhaitent interagir avec le composant, soit par le navigateur HTML lorsque l'utilisateur interagit avec l'interface.

3.4 Observateur d'interface HTML

Dès le début de l'exécution, l'observateur d'interface recherche à chaque cycle dans le code VDL du service les éléments décrivant l'interface utilisateur du service et produit une page web en HTML. Cette page, nommée $\#i.html$ où i est le numéro d'identification unique de la connexion, est donc modifiée à chaque cycle d'exécution. Elle est envoyée dès la connexion au client HTTP et est rechargée automatiquement par le client à intervalle régulier.[8]

[8]Cela est rendu possible à l'aide de la balise suivante dans l'en-tête (*head*) du document HTML :

```
<meta HTTP-EQUIV="Refresh" CONTENT=x>
```

où x est le délai (en secondes) entre deux rafraîchissements.

Notons que l'observateur d'interface est « paresseux » : il ne reconstruit la page $\#i.html$ que si des changements ont eu lieu dans les éléments décrivant l'interface du code XML du service.
Les éléments utilisés dans le code du service et reconnues par l'observateur HTML pour la définition de l'interface sont les suivants :

display pour commencer l'interface (l'observateur d'interface n'examine que le contenu de cet élément). L'attribut $refresh$ peut être utilisé pour fixer la période de rafraîchissement de la page par le client[9].

Dans un élément display, les balises de mise en forme HTML (changement de police, sections, paragraphes, listes, tableaux, *etc*) peuvent être utilisées, à condition d'écrire aussi les balises de fin facultatives. Elles sont interprétées comme du HTML. Par contre, les liens (**a**) et les entrées (**input**) sont ignorés.

textbox pour définir une zone de saisie : l'observateur d'interface produira une instruction HTML de la forme `<input type="text">`. L'élément $textbox$ doit être muni d'un attribut $name$ (pour définir l'attribut $name$ de l'input) et ne pas avoir de contenu.

textpass pour définir une zone de saisie masquée (l'observateur d'interface produira une instruction HTML de la forme `<input type="password">`).

Pour accéder dynamiquement au contenu d'une zone de saisies, le programmeur peut utiliser l'élément **getHTML** dont le contenu est une référence *XPath* à l'élément VDL qui décrit la zone de saisie dont il veut récupérer la valeur.

[9]Cette période est donc modifiable *au cours de l'exécution* du service.

Un exemple complet est donné section 4.2.

button pour définir un bouton qui peut contenir des sous-élément **text** pour définir le texte affiché sur le bouton et **event** pour définir l'événement externe qui sera envoyé lorsque l'utilisateur clique sur le bouton, c'est-à-dire le contenu du message SOAP. L'observateur d'interface produira alors une instruction HTML de la forme `<input type="button">` associée à un script client d'envoi du message SOAP. Par défaut, l'événement envoyé est un élément **clic** dont le contenu est l'adresse *XPath* du bouton dans la description XML du composant.

anchor pour définir des liens hypertextes, contenant nécessairement un sous-élément **text** pour définir le texte affiché dans le lien et pouvant éventuellement contenir un élément **event** pour définir l'événement externe envoyé lorsque l'utilisateur clique sur le lien. Comme pour les boutons, l'observateur d'interface produit un lien HTML qui appelle un script d'envoi de message SOAP.

Remarque. Lorsque l'interface contient des zones de saisie, l'observateur d'interface ne produit pas l'instruction HTML de rafraîchissement[10]. La page est alors rechargée uniquement lorsque le POST est effectué par le client, c'est-à-dire lorsque l'utilisateur clique sur un bouton ou un lien, ce qui déclenche l'envoi d'un événement externe. La page $\#i.html$ est alors remplacée par une page d'attente indiquant que les données sont en cours de traitement et mise à jour toutes les secondes. Lorsque l'interprète aura traité l'événement externe, une nouvelle interface sera produite et affichée dans le navigateur de l'utilisateur.
Le programmeur VDL doit s'assurer que les pages dynamiques (*i.e.* dans lesquelles les données évoluent de manière proactive) ne contiennent pas de zones de saisie.

3.5 Accès aux données de la base

Lorsque le service utilise une base de donnée, cette base est stockée dans des pages XML qui sont partagées par toutes les instances du service (et éventuellement par d'autres services). Les données contenues dans ces pages sont manipulées au sein du code VDL à l'aide de requêtes XQuery englobées dans un élément **xquery** muni d'un attribut $name$. Un exemple est donné section 4.3.
Les requêtes XQuery sont traitées au niveau de l'interprétation canonique VDL (*cf.* section 2.3). Par conséquent, elles peuvent être utilisées :
– Dans les actions VDL, effectuées par l'observateur d'exécution, pour modifier la base de données ;
– Dans la description (en VDL) de l'interface, afin d'afficher des données dans la page HTML de l'utilisateur (par exemple dans un tableau).

[10] En effet, les zones de saisies seraient alors vidées à chaque rafraîchissement !

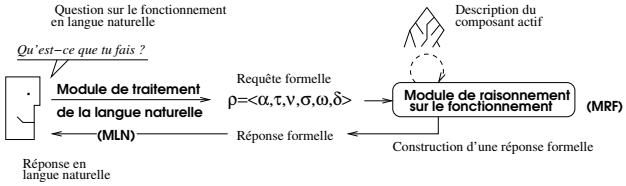

FIG. 4 – Traitement des questions de l'utilisateur.

3.6 Traitement des questions de l'utilisateur

Lors de la connexion initiale au service et de l'instanciation du code VDL, le système ouvre chez le client une deuxième page HTML contenant uniquement une « chatbox » en Java qui permet à l'utilisateur de poser des questions en langage naturel à propos des données manipulées par le service et surtout à propos des actions de manipulation, en s'appuyant sur les travaux effectués dans le cadre du projet *InterViews* [11, 7].
Cette applet Java communique avec le serveur de Traitement Automatique des Langues (TAL) du projet qui transforme les questions ou les commandes en requêtes formelles et les envoie au module de raisonnement sur le fonctionnement (MRF) qui traite les requêtes [7, 9]. Le système de TAL et le MRF utilisent la représentation des données en XML (sérialisées à chaque cycle). La réponse du MRF est une requête qui permet au système de TAL de produire une réponse en langage naturel renvoyée à l'applet et affichée à l'utilisateur.
La figure 4 illustre ce mécanisme, déjà mis en œuvre dans le cadre du projet *InterViews* dans un contexte opérationnel similaire.

3.7 Observateur de service web

L'observateur de service web génère les fichiers de déclaration de service web en WSDL (*grounding*) et DAML-S (*grounding*, *service profile* et *process*), de manière similaire à ce qui est fait pour l'interface HTML. Ces documents sont requis par l'utilisateur service ou agent pour savoir comment interagir avec le service VDL. Ils sont associés à l'instance du service crée lors de la connexion. De la même manière que pour l'interface, ce mécanisme peut être rendu transparent pour l'utilisateur.
Les documents DAML-S sont décrits dans le code même du service en VDL dans des éléments **damls** et **wsdl** qui sont utilisé par l'observateur pour générer les documents $grounding.wsdl$, $grounding.daml$, $process.daml$, $profile.daml$ et $service.daml$ à chaque cycle d'exécution. L'observateur de services est « paresseux » : il ne reconstruit les documents que lorsque le contenu des éléments $damls$ ou $wsdl$ est modifié. Chaque document est obtenu par concaténation des éléments pertinents pour ce document contenus dans les éléments $damls$

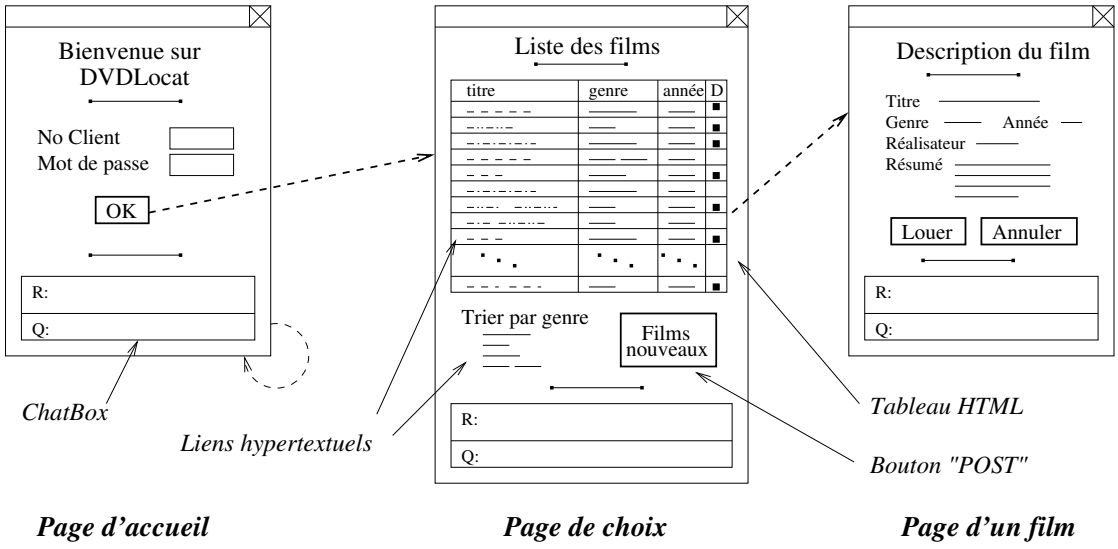

FIG. 5 – Un exemple de service : location de DVD.

ou $wsdl$ du code VDL (disséminés dans le code des différentes actions, comme nous allons le voir sur les exemples dans la section suivante).

Les instructions DAML-S doivent définir :

- Le « service grounding » DAML-S à l'aide d'éléments $grounding$ du schéma `daml/grounding.daml` ;
- Le « service profile » contenant la description en langage naturel de la fonctionnalité et la définition des paramètres du service (en particulier les entrées et sorties) ;
- Le « process model » du service qui définit les liens entre les différentes actions du process (par exemple lorsqu'une action VDL est composée de plusieurs sous-actions) et les propriétés DAML-S des entrées-sorties ;
- L'instance DAML-S du service qui regroupe, dans un élément $Service$, les descriptions du grounding, du profile et du process model définies ci-avant.

Les instructions WSDL doivent définir le « service grounding » WSDL, c'est-à-dire :

- Les messages SOAP en entrée et en sortie (éléments $message$) ;
- La définition des ports de communication ($portType$) ;
- Les « bindings » SOAP (éléments $binding$) contenant les opérations SOAP (éléments $operation$) ;
- La description du service (élément $service$) contenant la déclaration des ports (éléments $port$).

4 Un exemple complet

4.1 Présentation générale

Afin d'illustrer les concepts présentés dans la section précédente, nous considérons un service de location de DVD. Son interface est composée de plusieurs pages (*cf.* figure 5) :

- La page d'accueil et d'authentification du client ;
- La page principale, donnant la liste des films à louer (éventuellement triés par genre, etc) ;
- Une page spécifique pour chaque film donnant une description du film et contenant un bouton pour louer le film.

La figure 5 donne le schéma général de ce service.

4.2 L'interface HTML

L'interface de la page d'accueil sera définie en VDL par le code donné figure 6. Ainsi, l'événement envoyé lorsque l'utilisateur clique sur le bouton « OK » de la page de connexion sera de la forme :

```
<connexion>
  <login>...</login>
  <pwd>...</pwd>
</connexion>
```

où $login$ et pwd contiennent les valeurs entrées dans les zones de saisie correspondantes.

Le code HTML pour l'envoi de cet événement dans un message SOAP qui est généré par l'observateur d'interface à partir du code VDL et intégré dans l'interface utilisateur est donné sur la figure 7.

4.3 Accès à la base de données

Supposons que base de données du service DVD contienne une table des utilisateurs avec un champ $login$ et un champ pwd, représentée dans le fichier $user.xml$. L'action de connexion en VDL, dont l'objectif est de traiter les événements $connexion$, est donnée figure 8. Elle est munie de :

```
<display refresh=0>
  <html><h1>Bienvenue sur DVDLocat</h1>
  <table border=0><tr><td>Login :</td><td><textbox name="login"/></tr>
                  <tr><td>Mot de passe :</td><td><textpass name="pwd"/></td></tr>
  </table>
  <p><button name="ok"> <text>OK</text> <event>
    <connexion>
      <login><getHTML>//display/textbox[@name="login"]</getHTML></login>
      <pwd><getHTML>//display/textbox[@name="pwd"]</getHTML></pwd>
    </connexion></event></button></p></html>
</display>
```

FIG. 6 – Description de l'interface de connexion en VDL.

```
<input type="button" value="ok" onClick="send132()">
<script type="text/javascript"> send132() {
  var xmlhttp = new Active XObject("Msxml2.XMLHTTP.4.0");
  xmlHTTP.open("POST","http://www.DVDRent.com/VDLInterpreter");
  final = 'HTTP/1.1 200 OK
    [... content-type, etc ...]
    <SOAP:Envelope xmlns:SOAP=\"http://schemas.xmlsoap.org/soap\"/>
    <SOAP:Body> <connexion><login>' + document.login.value + '</login><pwd>'
    + document.pwd.value + '</pwd></connexion> </SOAP:Body></SOAP:Envelope>';
  xmlHTTP.send(final); } </script>
```

FIG. 7 – Code HTML associé au bouton « OK », généré par l'observateur d'interface.

- Deux actions élémentaires *put*. La première enregistre le login dans une variable du composant actif ; la seconde « charge » la page principale dans l'interface HTML.
- Une précondition de type $event$: l'action n'est effectuée que si le composant a bien reçu un événement $connexion$.
- Une précondition de type $guard$ qui vérifie, à l'aide d'une requête XML, si le login et le mot de passe figurent dans la base de données. L'action n'est effectuée que si le résultat de cette requête n'est pas nul.

En s'appuyant sur les algorithmes que nous avons proposés [7], le service est alors capable de répondre à des questions portant sur les actions de connexion, comme : « *Comment se connecter ?* », « *A quoi sert le bouton OK ?* », *etc*.

4.4 Déclarations de service web

L'observateur de service web produit, à chaque cycle d'exécution, à partir du code VDL figurant dans les élément $damls$ et $wsdl$ dans et hors des actions VDL, les documents de déclaration de l'interface du service. Par exemple, pour l'action de connexion, la description de ces fonctionnalités en DAML-S et WSDL est intégrée dans le code VDL de l'action « Connexion Reaction » dans les balises $damls$ et $wsdl$, comme indiqué sur la figure 8. Chaque action VDL doit contenir des éléments $damls$ et $wsdl$ pour déclarer ses fonctionnalités aux autres services.

Par exemple, pour l'action de connexion donnée figure 8, nous pouvons définir l'action de vérification du mot de passe comme un processus-atomique :

```
<daml:Class rdf:ID="LogIn">
  <daml:subClassOf rdf:resource=
    ".../Process.daml#AtomicProcess"/>
</daml:Class>
```

5 Conclusion et travaux en cours

Le modèle théorique de programmation de service que nous avons présenté ici permet de définir un service web comme un composant actif, capable d'interagir de manière proactive avec l'utilisateur humain ou avec d'autres services. La plate-forme de service web en VDL-XML sur un serveur HTTP est actuellement en cours de développement. Nous étudions la possibilité de réutiliser l'interprète VDL existant, implémenté en Java [8] et d'y adjoindre un observateur d'interface HTML et un observateur de services. Nous projetons ensuite d'implémenter ces observateurs et l'architecture logicielle qui a été présentée.

Nous pensons que le modèle VDL que nous avons présenté ici ouvre de nouvelles voies dans le domaine de la composition de service. En effet, le service que nous avons défini peut être découvert, invoqué par et composé avec d'autres services puisqu'il génère toutes les informations DAML-S nécessaires.[11] Mais il est aussi possible de *programmer en VDL* les mécanismes de composition de service : les ac-

[11] En fait, il subsiste toujours un problème d'ontologie : deux services devant être composés doivent s'accorder sur l'ontologie (le vocabulaire) qu'ils utilisent. Ce problème, très étudié dans la communauté, n'est toujours pas résolu aujourd'hui.

```
<action>  <name>Connection reaction</name>
  <damls> ... définition du service profile et du process ... </damls>
  <wsdl> ... définition du grounding (binding SOAP, etc) ... </wsdl>
  <event><connexion/></event>
  <guard><not><equals><xquery name="checklog">
      for $p in document("users.xml")/users/user_tuple
      where $p/login = event/connexion/login and $p/pwd = event/connexion/pwd
      return $p/userid
    </xquery><null/></equals></not></guard>
  <put><path>//variables/login</path><eventget><login></eventget></put>
  <put><path>//interface</path><get>//Main_screen</get></put>
</action>
```

FIG. 8 – Action de connexion en VDL et XQuery.

tions de composition et leur déclaration en DAML-S sont alors *intégrés dans une même représentation*.

Par exemple, dans notre service de location de DVD, l'authentification fait partie intégrante du service : le code procédural associé à ce « process » atomique est décrit en VDL et directement relié à la suite du service. Mais il est possible de séparer ce process du reste du service et d'en faire un service autonome. Le service de location doit alors contenir des instructions VDL pour rechercher le service de connexion dans un annuaire UDDI, l'invoquer et le composer avec le reste des opérations du service, définies en VDL.

Comme pour la manipulation de bases de données, l'écriture de ce code d'invocation et de composition en VDL relève simplement de l'interprétation d'éléments spécifiques à l'appel de services DAML-S.

Notre objectif à court terme est d'intégrer *dans l'observateur d'exécution* l'interprétation des opérations d'invocation de services DAML-S de manière à pouvoir implémenter en VDL des services actifs capables d'interagir dynamiquement avec d'autres services. De plus, ces services actifs seraient capables de raisonner sur ce qu'ils font et d'interagir avec l'utilisateur humain pour répondre à des questions sur son fonctionement, en s'appuyant sur les algorithmes que nous avons proposé dans [7] et [9]. Ainsi, le service devient capable d'expliquer ce qu'il se passe en lui et autour de lui : pourquoi tel service a été choisi, comment il a été appelé, comment l'interaction s'est effectuée, *etc*.

Enfin, parce que le composant a accès en cours d'exécution à ses propres instructions de composition de service (en VDL), il devient possible d'adapter le service de manière automatique et *au cours de l'exécution* aux fonctionnalités des services ou des agents avec lequel il interagit.

Références

[1] T. Andreasen, J.F. Nilsson, and H.E. Thomsen. Ontology-Based Querying. In *Proc. 4th Intl. Conf. on Flexible Query Answering Systems (FQAS'2000)*, pages 15–26, 2000.

[2] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl). http ://www.w3.org/TR/wsdl, 2001.

[3] DAML-S Coalition. *DAML-S : Web Service Description for the Semantic Web*, 2002.

[4] D. Fensel and C. Bussler. The Web Services Modeling Framework WSMF. In *1st meeting of the "Semantic Web enabled Web Services workgroup"*, 2002.

[5] Z. Guessoum and J.P. Briot. From Active Objects to Autonomous Agents. *IEEE Concurrency*, 7(3) :68–76, 1999.

[6] J. Meseguer. Rewriting Logic and Maude : Concepts and Applications. In *Proc. RTA 2000*, pages 1–26, 2000.

[7] N. Sabouret. A model of requests about actions for active components in the semantic web. In *Proc. STAIRS 2002*, pages 11–20, 2002.

[8] N. Sabouret. *Étude de modèles de représentation, de requêtes et de raisonnement sur le fonctionnement des composants actifs pour l'interaction homme-machine*. PhD thesis, Université Paris-Sud, 2002.

[9] N. Sabouret and J.P. Sansonnet. Traitement de questions naturelles sur le fonctionnement. In *Proc. RFIA 2002*, volume 2, pages 643–652, 2002.

[10] N. Sabouret and J.P. Sansonnet. Un langage de description de composants actifs pour le web sémantique. *Revue RI^3*, page To appear, 2003.

[11] J.P. Sansonnet. Description Scientifique du Projet InterViews — The InterViews Project. Technical Report 99-01, LIMSI-CNRS, 1999.