RNTL FAROS
Composition de contrats pour la Fiabilité d'ARchitectures Orientées Services

Livrable F-3.1
Coordonnateur : Lionel SEINTURIER

Identification des modalités de prise en charge des contrats pour chaque plate-forme cible

Projet FAROS

Juin 2007 - Version 1.0

Projet RNTL FAROS : `http://www.lifl.fr/faros`

CHAPITRE 3

WComp

Coordonnateur : Jean-Yves Tigli.
Rédacteurs : Daniel Cheung, Vincent Hourdin, Stéphane Lavirotte, Anne-Marie Pinna Dery.

Ce chapitre présente les modalités de prise en charge de différentes formes de contrats dans la plate-forme WComp/WS UPnP pour FAROS.
Cette plate-forme permet l'adaptation dynamique des applications en informatique ambiante reposant sur une infrastructure composée à la fois de *Web Services* et de *Web Services pour dispositifs*.
Nous traduisons ici "input/ouput devices" par *dispositif d'entrée/sortie* ou *dispositif*. Les dispositifs couvrent le large spectre des équipements permettant l'interaction contrôlée de la machine et donc du logiciel avec le monde réel. Ils ne se limitent pas à la seule notion de périphérique. Il peut s'agir de capteurs, d'effecteurs de natures diverses que l'on rencontre dans de multiples domaines d'applications telles que la Domotique, le "Wearable Computing", les réseaux de capteurs, etc. La notion de *Web Services pour dispositifs* regroupe alors les approches basées sur des standards adaptés de Web Services pour l'accès à ces dispositifs, tels qu'UPnP et DPWS que nous détaillerons plus loin.
Pour illustrer cette problématique, WComp est au cœur d'un environnement expérimental original : une plate-forme d'étude des usages des équipements informatiques mobiles en environnement simulé, appelé « Ubiquarium[1] Informatique ».
La section 3.1 présente la plate-forme WComp et ses évolutions pour FAROS ainsi que l'infrastructure sous-jacente constituée par des Web Services classiques et des Web Services pour Dispositif s'exécutant sur des serveurs UPnP. La section 3.2 détaille la notion d'aspect d'assemblage et sa mise en œuvre. La section 3.3 énumére les extensions de la plate-forme et les principes pour la mise en œuvre de contrats. La section 3.4 présente deux exemples de mise en œuvre de contrat sur la plate-forme WComp utilisant des aspects d'assemblage. Ce chapitre est conclu par la section 3.5.

3.1 Description de la plate-forme WComp/WS UPnP

Avec la multiplication des terminaux mobiles et des objets communicants dans notre vie quotidienne, nous observons l'émergence d'une informatique mobile et ambiante dont il est essentiel d'anticiper l'impact sur le développement d'applications logicielles et l'interaction avec l'humain. La rupture par rapport à l'informatique traditionnelle tient à la nécessité d'adapter les services des applications à la mobilité dans le milieu physique (des dispositifs ou de l'utilisateur) et à la dynamique des interconnexions. Cette nécessité d'adaptation des applications informatiques [TCFWLR06] est due au fait qu'elles sont déployées sur un ensemble de dispositifs matériels variés et hétérogènes.
WComp est à l'origine une plate-forme à composants pour le développement rapide de prototypes d'applications multi-dispositifs qui doivent évoluer avec leur environnement d'exécution [CFWTLR07].

[1] du Latin Ubique, en toute chose et tout être, avec le suffixe rium signifiant lieu ou structure. Donc Ubiquarium Informatique : « lieu ou structure dans laquelle l'informatique est en toute chose et tout être »

3.1.1 Infrastructure WS UPnP de la plate-forme WComp

La plate-forme WComp repose donc sur un *environnement d'exécution* logiciel et matériel évoluant dynamiquement. Nous définissons cet environnement par un ensemble de *ressources*, comme autant d'entités logicielles/physiques dont l'apparition et la disparition ne sont pas pilotées mais subies par l'application.
Plus spécifiquement dans le cadre de WComp pour FAROS nous limitons les ressources à l'ensemble des services accessibles et des dispositifs utilisables par la plate-forme que nous appellerons *infrastructure* de la plate-forme. Afin de ne pas être limité dans notre approche par des problèmes d'interopérabilité entre différentes approches technologiques d'accès aux services [HLT06], nous concentrons nos travaux sur l'utilisation de technologies Web et donc de Web Services standards et de Web Services pour dispositifs [HLT07] (dont une première mise en œuvre est apparue avec le standard UPnP).

Des Web Services classiques aux Web Services pour dispositifs

Les Web Services ont été décrits dans le chapitre 1 de l'état de l'art. Ils constituent une approche pour mettre en œuvre le paradigme de service dans un contexte technologique facilitant l'interopérabilité des services indépendamment des plates-formes sous-jacentes.
Utiliser cette approche pour accéder à des dispositifs permet de gérer l'apparition ou de la disparition de dispositifs comme l'arrivée ou le départ d'un service. Deux extensions majeures sont alors nécessaires :
– la *recherche et découverte dynamique* de nouveaux services pour dispositifs dans l'environnement proche de l'utilisateur,
– un mode de communication par *événements* garantissant un minimum de réactivité aux variations des entrées/sorties des dispositifs.

Les premiers travaux en ce sens ont été initiés par le consortium UPnP (*Universal Plug and Play*) en 1999. Depuis le W3C a publié un certain nombre de soumissions de recommandations (ex. [WS-b], [WS-a]) pour introduire le standard DPWS (*Device Profile for Web Services*).

UPnP est la première implémentation des Web Services pour Dispositifs (*Web Services for Devices*), bien qu'elle ne se définisse pas ainsi. Ce standard a été créé en 1999 par Microsoft et Intel, à partir de technologies en partie existantes et fiables comme IP, UDP et TCP, HTTP, XML, SOAP, GENA (General Event Notification Architecture), HTTPU et HTTPMU (HTTP sur UDP et MulticastUDP). Elle a ensuite été développée par le forum UPnP que plusieurs centaines d'entreprises ont rejoint.
Le premier but d'UPnP [HLT07] était l'extension des notions de Plug'n'Play à tout dispositif du réseau local. Les choix technologiques qui ont été faits pour arriver à ce but ont permis de l'utiliser en tant que Web Service pour dispositif. Cinq fonctionnalités sont supportées par UPnP :
– Recherche et découverte : les dispositifs (les fournisseurs de services, appelés serveurs ou *devices* en UPnP) sont découvrables sur réseau local passivement ou activement, grâce à des messages diffusés en UDP.
– Description : les serveurs peuvent fournir un ou plusieurs services, qui implémentent des *méthodes* et fournissent plusieurs sources d'événements par des *variables événementielles* (*evented variables*). Ces capacités sont fixées et décrites dans des fichiers de description des serveurs et services UPnP, au format spécifique à UPnP, le SCPD (Service Control Protocol Description).
– Invocations synchrones : les appels de méthode par SOAP sont identiques aux appels de méthode des Web Services classiques.

– Souscription aux événements : s'abonner aux variables événementielles permet de recevoir des notifications de changement de leur valeur.
– Présentation : en plus des fonctionnalités de Web Service pour dispositif, UPnP permet la présentation de pages html sur un serveur HTTP embarqué. Cette fonctionnalité d'UPnP est la plus utilisée par les industriels, qui s'en servent essentiellement pour fournir une page web de configuration du dispositif ou de téléchargement de pilote pour utiliser le dispositif UPnP avec des moyens plus conventionnels que les appels de méthodes.

UPnP remplit donc toutes les conditions pour être classé parmi les Web Services pour Dispositifs. Cependant, l'utilisation de fichiers de descriptions et d'événements de syntaxe spéciale, et l'absence de gestion de sécurité telle que l'authentification ou le cryptage révèlent certaines lacunes dans cette première implémentation, résolus dans UPnP version 2 nommé DPWS.

DPWS *Device Profile for Web Service*, définit un ensemble minimal de spécifications pour la recherche, la découverte, la description, la transmission de messages, et la diffusion d'événements liés à un dispositif. DPWS est en ce sens similaire à l'approche UPnP, dont il est en fait la deuxième version, mais est entièrement compatible avec les standards des Web Services en incluant de nombreuses extensions basées sur WSDL et SOAP pour l'intégration de services pour dispositifs dans des approches SOA. Dans ces extensions, nous retrouvons :
– WS-MetadataExchange qui définit le format des fichiers de description des services. En plus des informations présentes dans les descriptions des devices et services UPnP, apparaissent les traductions de certaines informations dans plusieurs langues, et diverses URL pointant vers les adresses de retour des invocations, de présentation, d'accès aux autres fonctionnalités, etc.
– WS-Discovery qui définit les normes de recherche et découverte. En plus de se baser sur WSDL et SOAP, les améliorations par rapport à UPnP sont la faculté d'utiliser de façon transparente un annuaire local de services et l'ajout de critères de recherche supplémentaires, avec l'utilisation d'une syntaxe LDAP qui supporte des filtres comportant plusieurs conditions logiques.
– WS-Eventing qui définit les normes pour la souscription et l'envoi d'événements. L'envoi d'un événement peut se faire suivant plusieurs protocoles, le plus classique étant le push, semblable aux événenements d'UPnP. Le fournisseur d'événements fournit une fonctionnalité de filtre des messages envoyés, qui peut être configuré lors de la souscription par le consommateur. L'envoi d'un événement n'oblige plus à envoyer les valeurs de toutes les variables d'un service, comme c'était le cas avec UPnP. Il est aussi possible de déporter la gestion des souscriptions, hors de l'entité qui génère les événements.
– WS-Security qui définit les normes d'authentification, de signature et de cryptage des données transmises par le serveur. C'est un apport important par rapport à UPnP, qui permettra d'utiliser des services Web pour dispositifs pour lesquels l'utilisation par une personne mal intentionnée serait critique.

Environnement expérimental de WComp : Ubiquarium

L'Environnement expérimental de WComp, appelé *Ubiquarium* est constitué de divers dispositifs, comme autant de services découvrables et composables dynamiquement par WComp. Ces dispositifs peuvent être à la fois des dispositifs virtuels (objets d'une scène 3D dans laquelle l'utilisateur est immergé), et des dispositifs réels portés sur lui ou présents dans son environnement.
Un tel environnement permet l'évaluation des nouvelles applications de l'informatique mobile et ambiante tels que les usages des ordinateurs portés ou « wearable computers ». C'est un cadre idéal pour le prototypage de nouvelles applications pour les utilisateurs mobiles. L'infrastructure de l'Ubiquarium repose sur la plate-forme WComp. Une partie de ces travaux

serviront à enrichir l'interactivité avec l'utilisateur dans la plate-forme SEDUITE de FAROS.

3.1.2 La plate-forme WComp

La plate-forme WComp a donc pour vocation de gérer à la fois une application comme un assemblage de composants logiciels et la dépendance de certains composants par rapport aux ressources de l'infrastructure.

Une plate-forme à événements : La programmation par événements simplifie la réalisation d'applications faiblement couplées. Dans WComp, les connecteurs ou composants de liaison sont basés sur la diffusion ou propagation d'événements conformément au modèle Bean4WComp [CFWBFT+06]. Chaque composant émet des événements pour informer les autres d'un changement d'état interne ou pour invoquer une méthode d'un autre composant. Les composants réagissent à la réception d'un événement, par exemple en émettant eux aussi des événements.

Différents types de composants : La plate-forme WComp gère des assemblages de composants logiciels de deux types :
- *composants mixtes* : ce sont des composants logiciels présents dans la plate-forme uniquement si les *ressources* (telles que définies dans la section 3.1.1) qu'ils utilisent sont accessibles. Ils peuvent apparaître ou disparaître au gré de l'apparition et de la disparition des ressources dans l'infrastructure de la plate-forme décrite précédemment. Cette catégorie peut être divisée en deux sous-catégories :
  - ceux qui ont une interaction directe avec les ressources
  - les clients de services (c'est-à-dire les composants proxy)
- *composants logiciels* : ceux sont les composants qui ne dépendent pas de la variation des ressources.

Ces composants logiciels sont exécutables dans un environnement d'exécution donné si celui-ci possède de manière permanente les ressources logicielles et matérielles pour l'exécuter. Dans le cas de l'Ubiquarium et des applications qui nous intéressent dans FAROS, les ressources sont limitées aux services et dispositifs accessibles. Les composants mixtes sont alors les composants logiciels proxy de Web Services et Web Services pour Dispositifs type UPnP. La modification dynamique de l'infrastructure par apparition ou disparition des services web pour dispositifs qui s'y trouvent provoque la génération/création ou destruction du composant proxy correspondant dans l'assemblage WComp de l'application. Cette modification minimale de l'assemblage pouvant induire d'autres modifications et donc l'adaptation de l'application par composition.

WComp : Une approche multi-design du développement par composition

La gestion et les modifications des assemblages de composants WComp peuvent utiliser différents modèles. L'architecture de WComp s'organise donc autour de *container*s et de *designers* [CFWTLR06]. L'objectif des *containers* est de prendre en charge dynamiquement et à l'exécution la gestion de comportements tels que l'instanciation, la désignation, la destruction de composants logiciels fonctionnels et de liaisons. Les *designers* permettent de manipuler ces modèles de l'application en utilisant les formalismes adaptés. Nous en avons trois :
- un *designer* graphique d'architecture Bean4WComp,
- un *designer* de composants mixtes, i.e. de proxy de Web Services et de Web Services pour Dispositifs (SOAPProxyWizard et UPnPProxyWizard),
- un *designer* d'aspects d'assemblage.

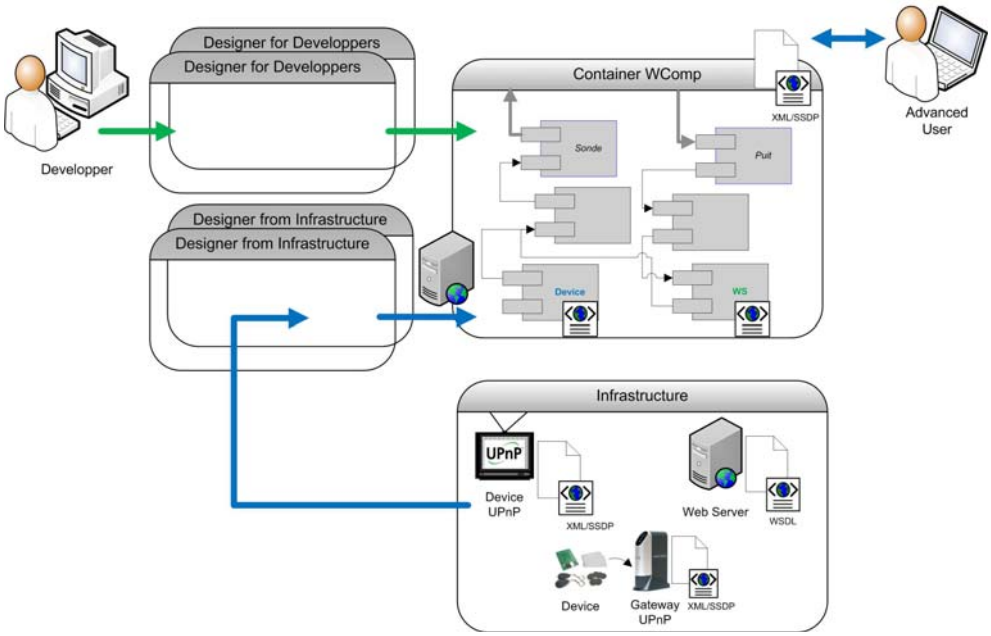

FIG. 3.1: Approche multi-design WComp

Le *designer* **graphique d'architecture Bean4WComp** permet par exemple de composer manuellement des assemblages de composants à partir d'une représentation graphique des flots d'événements. Il est particulièrement adapté à la description de l'application.
Ce *designer* est à rapprocher de la conception ADL d'une application. L'application est construite en assemblant des composants sur étagères et en les reliant par diffusion d'événements. Cette méthode de conception prend toute son ampleur pour une conception rapide d'application mais devient limitée lorsque la fréquence d'apparition et de disparition augmente. [CFWTLR06].

Les *designers* suivants permettent de gérer l'apparition/disparition des composants proxy de Web Services pour Dispositifs dans le *container*.

Le *designer* **de composants mixtes pour web services** génère et instancie automatiquement les composants proxy des web services standard adressés. La description du web service au format WSDL est utilisée pour générer le composant proxy WComp correspondant (figure 3.2).

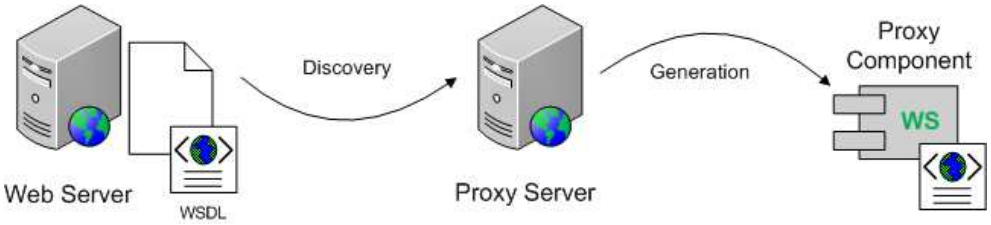

FIG. 3.2: Composant proxy pour web service

Le *designer* **de composants mixtes UPnP** génère et instancie automatiquement les composants proxy de devices UPnP. Il gère les variations de l'infrastructure sans intervention directe de l'utilisateur. Il ne traite pas de la modification de l'assemblage de composants mais uniquement de leur instanciation/désinstanciation, autrement dit, de leur ajout et de leur retrait de l'assemblage. Il est cependant configurable pour ne pas prendre en compte certains dispositifs dans l'assemblage de composants. L'utilisateur ne peut agir sur le comportement de ce *designer* qu'en modifiant l'état réel des dispositifs physiques.

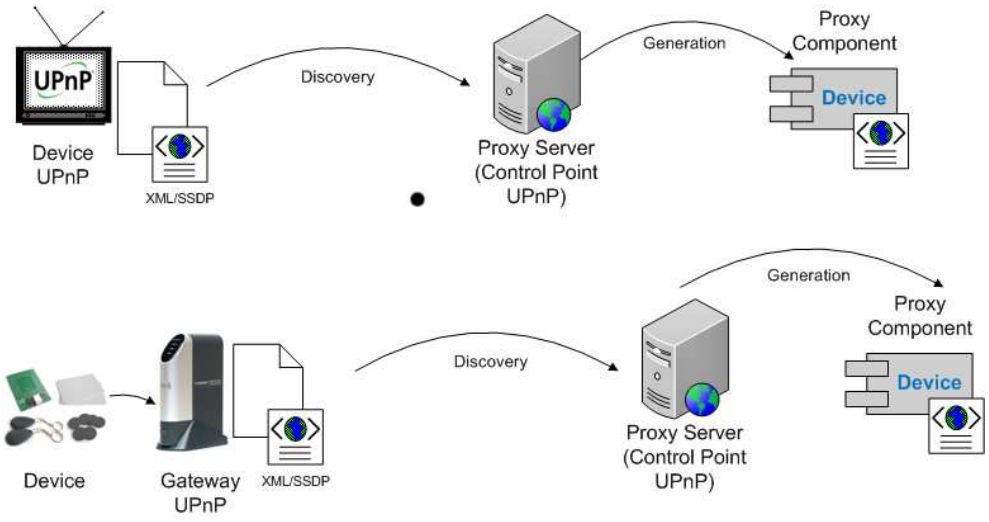

FIG. 3.3: Composants proxy UPnP

Les *designers* **pour Aspects d'Assemblage** accompagnent le premier *designer* en résolvant le problème de l'adaptation des assemblages de composants en permettant de greffer des schémas d'assemblage en réponse à la variation de l'infrastructure sous-jacente [TCFWLR06]. Nous détaillerons les principes de ce *designer* au cœur de la mise en œuvre des contrats dans la section 3.2.

Évolution de la plate-forme WComp pour l'orchestration de services

Afin que la plate-forme WComp puisse être utilisée pour l'orchestration de services (quel que soit son type), une extension de WComp a été développée pour permettre d'intéragir avec le *container* au travers un Web Service pour Dispositifs.
Cette évolution a nécessité l'introduction de composants spécifiques pour l'interception et le déclenchement d'événements et d'appels de méthodes dans l'assemblage géré par le *container*. Nous les appelerons *composants sondes*.

Les composants sondes sont de deux types :
– les composants sources d'événements pour le Web Service pour Dispositifs du *container*,
– les composants puits d'événements pour le Web Service pour Dispositifs du *container*.

Les composants sources sont de deux types :
– les composants d'interception d'événement,
– les composants d'interception d'appel de méthodes.

Les composants puits sont de deux types :

– les composants de déclenchement d'événements,
– les composants de déclenchement d'appel de méthode.

Ces composants permettent aux *designers* d'intervenir sur les flots d'événements de l'assemblage de l'application. Ils correspondent à autant d'événements et d'opérations invocables sur le *container* au travers du Web Service correspondant.
Les autres événements et opérations correspondent aux modifications de l'assemblage et au cycle de vie de l'application gérée par le *container*, comme les événenements de démarrage de l'application et des modifications élémentaires de l'assemblage.
Les opérations, quant à elles, sont très proches d'un ADL et des modifications élémentaires de l'assemblage de l'application.

```
add_component Type (Id) [ou simplement add]
remove_component Id [ou simplement remove]
link Id Source_Event Target_Method (Params)
unlink Id
```

Le container présente donc une interface de type Web Service pour dispositif qui permet à la fois d'interagir avec l'assemblage de ses composants et le flot de ses événements.

3.2 Aspect d'assemblage

3.2.1 Concepts

Le concept d'aspect d'assemblage se définit en trois volets :

1. Le *point de coupe* ou l'expression des points de jontion dans l'assemblage où l'aspect d'assemblage est introduit,
2. Le *greffon* ou la représentation de l'assemblage qu'on introduit (dans notre cas il s'agit d'un schéma d'assemblage)
3. Le *tisseur* ou la méthode pour composer les assemblages à introduire dans l'assemblage existant.

Le premier volet identifie les points d'ancrage de la modification d'assemblage produite par le schéma d'assemblage de l'aspect. Le deuxième volet décrit l'assemblage à introduire, en utilisant des opérateurs bien spécifiques au schéma d'assemblage de l'aspect. Pour un domaine particulier, il n'est pas utile de spécifier le troisième volet qui est fixé une fois pour toute. Ce volet consiste à traduire les spécifications décrites dans le second volet en un ensemble de modifications élémentaires qui vont entrainer une modification de l'assemblage d'origine.

Les points de coupe des Aspects d'Assemblage

Les points de coupe de l'aspect décrivent par une expression régulière les points de jonction où le schéma d'assemblage doit se positionner. Ces expressions régulières portent sur les identifiants des instances de type de composant et sur leurs ports. Un point de coupe est donc ici représenté par une expression régulière. Un point de jonction est un des identifiants de l'assemblage désigné par l'expression régulière.

```
(<composant>|<événement>|<méthode>) = <expression régulière>
```

Opérateurs	Description
;	exprime une séquence
||	exprime un ordre indifférent entre les comportements
if ... else ...	exprime une condition
call	désigne l'événement ou l'appel du point de jonction
delegate	désigne l'émission d'un événement ou un appel
^	exprime l'émission d'un événement

TAB. 3.1: Principaux opérateurs du langage ISL4WComp

Les greffons des Aspects d'Assemblage

Les greffons des Aspects d'Assemblage reposent sur le schéma d'assemblage correspondant. Nous nous appuyons dans la version actuelle de WComp sur une description des schémas d'assemblage basée sur le langage de spécification d'interactions ISL (Berger, 2001). ISL (Interaction Specification Language) est un langage de description de schémas d'interactions entre des composants. ISL4WComp étend ISL pour la prise en compte des interactions basées sur des événements. Ces langages ont la propriété d'être composables c'est-à-dire que plusieurs schémas peuvent se composer et fusionner en un seul schéma combinant leur comportement. Nous utilisons ces propriétés dans les aspects d'assemblage, notamment la symétrie - c'est-à-dire que l'ordre d'application des aspects n'influence pas le résultat de leur composition - démontrée par Berger. Au sein d'ISL4WComp, nous avons défini deux façons de modifier le comportement d'une application :
– d'une part, sur un appel de méthode,
– d'autre part, sur la diffusion d'un événement.
Dans tous les cas, on y associe un nouveau comportement programmé avec les opérateurs du langage (cf. Tableau 3.1).

```
<point de jonction> : <composant>.(<événement>|<méthode>)
aspect <nom> (<composants>, ...) {
  <point de jonction> { <comportement programmé ISL4WComp> }
}
```

Un aspect d'assemblage repose donc sur des ensembles de règles. Une règle porte sur des points de jonctions :
– émission d'un événement
– appel de méthode.
Certains mots-clés (cf. Tableau 3.1) du langage comme call et delegate permettent de contrôler la manière dont les schéma d'assemblage des aspects d'assemblage vont se composer. Nous pouvons les rapprocher des mots-clés comme before, around et after d'AspectJ. La fusion des schémas d'assemblage correspond à la mise en œuvre des travaux formels sur la composition des règles pour ISL4WComp. La définition d'aspects d'assemblage est donc un moyen de représenter des modifications d'assemblages de composants en y apportant la possibilité de les fusionner automatiquement. Des exemples de composition d'aspects d'assemblage peuvent être trouvés dans [CFWBFT+06]. Nous décrivons dans la section ci-après l'écriture d'un aspect.

Les tisseurs des Aspects d'Assemblage

Notre technique d'application des aspects d'assemblage est constituée d'un mécanisme de tissage d'aspects dans le domaine de la programmation logicielle orientée aspect AOP appliquée aux composants logiciels [Sei06] pour modifier l'assemblage qui implémente l'application. Les aspects d'assemblage sont décrits en respectant les interfaces des composants

WComp. La composition des aspects est réifiée au niveau d'un autre modèle de l'application : le modèle ISL4WComp. Le calcul de leur tissage et l'évolution dynamique de l'assemblage de composants sont pris en charge au niveau de ce modèle. Le résultat des calculs est ensuite projeté en termes de réassemblages des composants WComp. Il s'agit d'une des particularités de notre approche sur la fusion des aspects d'assemblage [CFWBFT+06].

3.2.2 Le designer par Aspects d'Assemblage

Le schéma général fonctionnel du *designer* d'Aspect d'Assemblage est décrit dans la figure 3.4.

Un aspect d'assemblage permet en effet de décrire simultanément des points de coupe et des schémas d'assemblage associés en lieu et place des greffons dans le cas de langages d'aspect. Les points de coupe permettent alors de définir l'ensemble des points de jonction ou d'application des schémas d'assemblages concernés.

Nous allons voir le *designer* qui a été mis en place lors de cette recherche : le *designer* ISL4WComp.

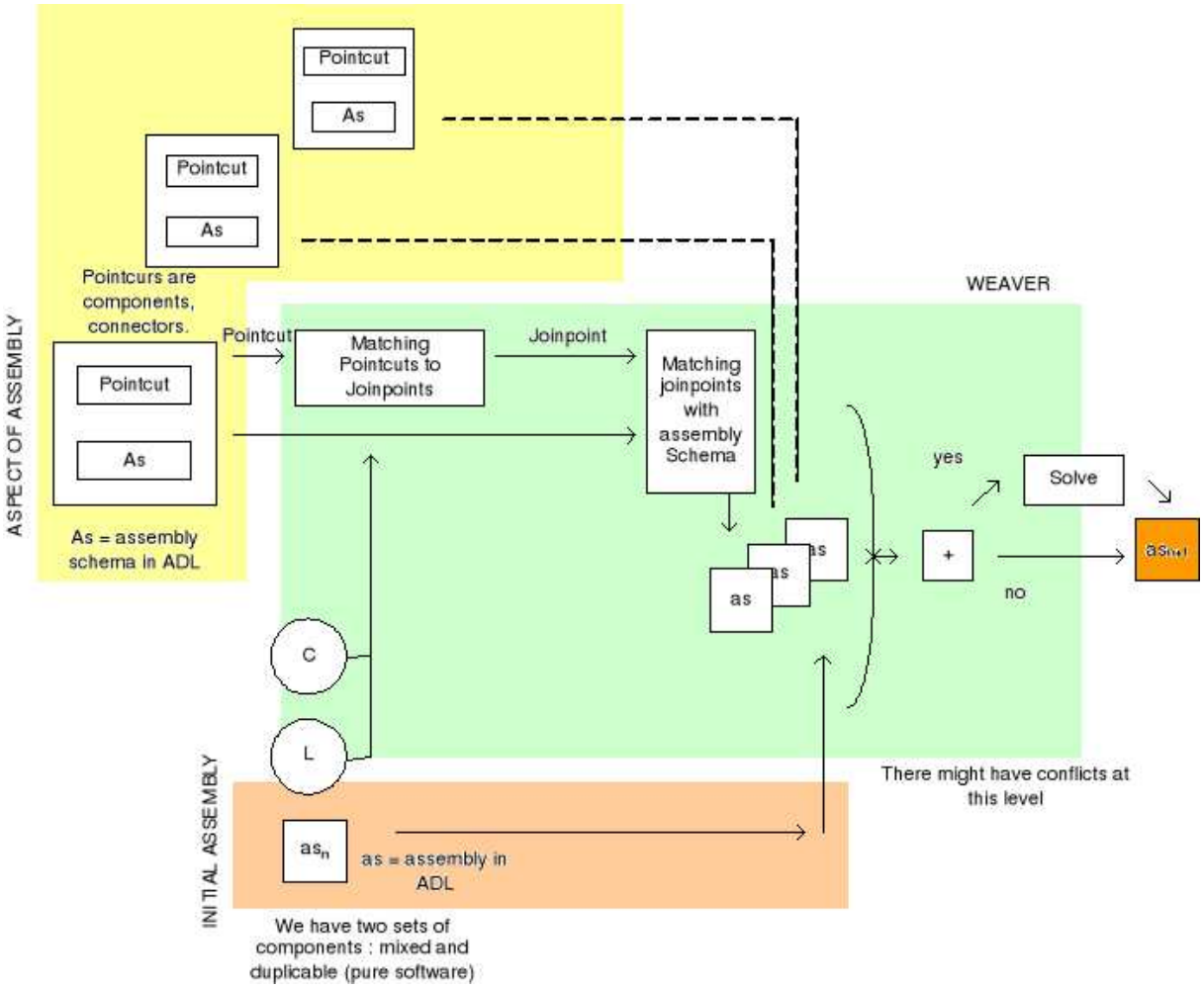

FIG. 3.4: Schéma général de mise en œuvre des Aspects d'Assemblage

3.3 Principes des extensions de WComp/WS UPnP pour la mise en œuvre de contrats

Les évolutions de la plate-forme WComp (WComp/WS UPnP) pour l'orchestation de services présentées dans la section 3.1.2 permettent une recomposition dynamique d'assemblage des composants d'une application et permettent d'instrumenter ainsi l'application comme décrit dans la section 3.1.2. Les sous-sections suivantes détaillent les mécanismes de la plate-forme qui permettent cette instrumentation, les informations observables à partir de ces mécanismes, et finalement, les comportements que l'on peut mettre en œuvre en réaction à une violation de contrat.

3.3.1 Instrumentation

Les évolutions de la plate-forme WComp pour l'orchestation de services présentée dans la section 3.1.2 permettent de créer un composant proxy sur ce *container* dans une autre *container* WComp. L'interface du composant contient alors méthodes et événements qui permettent d'interagir avec l'application du premier *container*. Une présentation de l'interface du web service pour dispositif correspondant se trouve dans la figure 3.5.

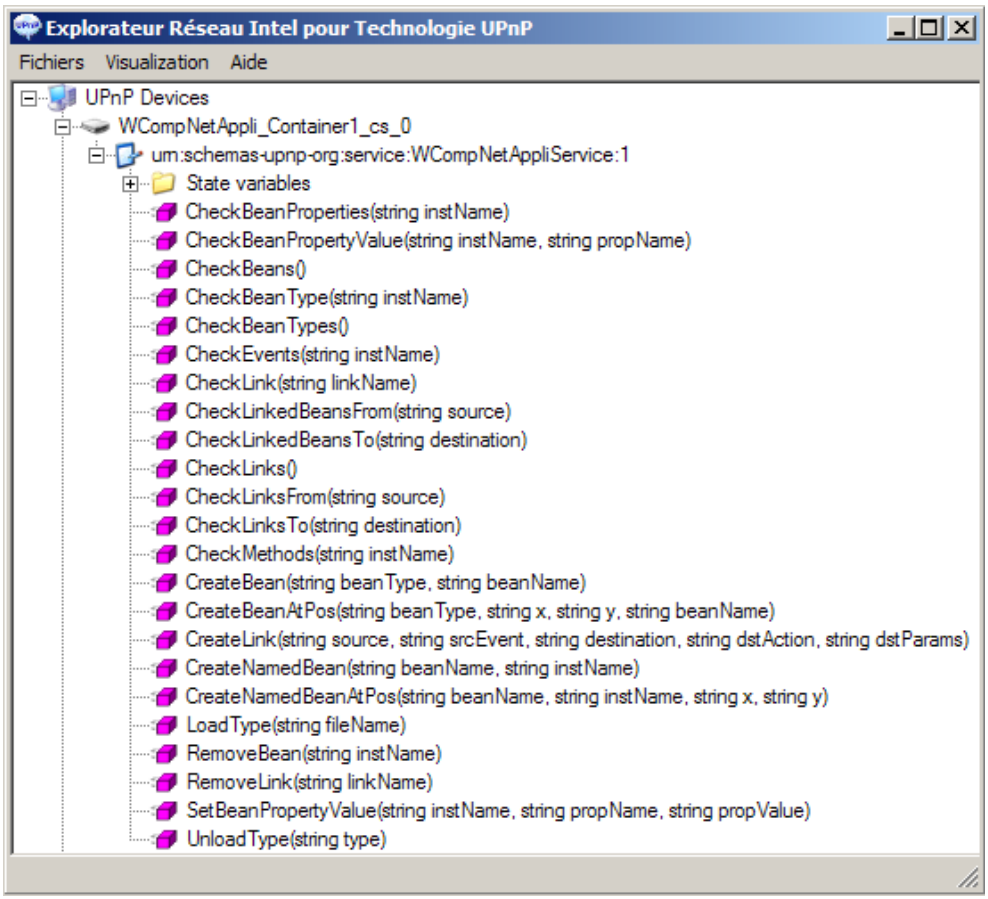

FIG. 3.5: Interface UPnP du *container* WComp

Elle permet d'intercepter les événements émis comme les appels de méthodes du *container*. Il s'agit :

– d'événements et méthodes exportées de l'assemblage (ex : événements émis ou méthodes appelées, de l'assemblage, interceptées par les composants sondes)

– d'événements liés au cyle de vie de l'application gérée par le *container* (ex : événement de bootstrap)
– d'événements et méthodes de manipulation de l'assemblage (ex : méthodes directement inspirées de l'ADL de la plate-forme, mais aussi événenents émis sur l'apparition/disparition de composants mixtes tels que définis dans le paragraphe 3.1.2).

Les modifications dynamiques de l'assemblage de composant dans WComp peuvent être directes ou en utilisant le designer par aspects d'assemblage (section 3.2).
Dans le premier cas l'instrumentation peut se faire par ajout/retrait d'un composant spécifique, généré et inséré dans l'assemblage pour intercepter les émissions d'événements et les appels de méthodes du flot d'événement.
Dans le second cas, nous utilisons un ensemble de composants prédéfinis dans des aspects d'assemblage comme autant de motifs d'instrumentation prédéfinis. Les aspects d'assemblage permettent alors une instrumentation de l'assemblage de composants, sans génération de composant a posteriori, par principe. Les interceptions multiples peuvent être ainsi mis en œuvre par sélection et tissage d'aspects d'assemblage et modifications résultantes de l'application.

3.3.2 Observations

Les informations observables reposent sur les mécanismes d'instrumentation précités. Nous pouvons donc distinguer :
– les informations observables des flots d'événements, en interne, par modification de l'assemblage de l'application,
– les informations observables des flots d'événements, en externe, au travers le *container* après la pose de composants sondes dans l'assemblage,
– les informations observables liées aux modifications de l'assemblage et au cycle de vie de l'application, gérés par le *container*.

Ces informations observables permettent la mise en œuvre de trois des quatre types de contrats de la classification de [BJPW99], même si le mécanisme de vérification des contrats en lui-même reste à la charge du développeur.
Les contrats comportementaux, de synchronisation ou de qualité de service, peuvent être mise en place grâce à l'introduction de nouveaux composants dans l'assemblage, ou par la mise en place d'un service externe dialoguant avec le *container*.

3.3.3 Réactions à la violation

Nous distinguerons plusieurs niveaux de réaction à la violation de contrat selon les conséquences de la violation sur l'application.
Il pourra s'agir
– d'une réaction locale dans la gestion des flots d'événements au travers des composants introduit dans l'assemblage pour la mise en œuvre du contrat.
– de modifications de l'assemblage lui même par ajout/retrait de composants, après selection de nouveaux aspects d'assemblage ou par insertion directe de composants dans l'assemblage.

3.4 Mise en œuvre

Nous allons présenter dans ce pararaphe trois exemples de mise en œuvre de contrat de qualité de service.
Le premier concerne un composant proxy de web services standard. Il s'agit de vérifier la durée d'exécution d'une requête (Timeout). Le second plus spécifique à notre approche illustre

la mise en œuvre d'un tel contrat sur un web service pour dispositif. Il s'agit alors de vérifier la période entre deux événements du web service consécutifs (Timeout).
Dans WComp, nous disposons de deux possibilités pour adapter une application : on peut soit effectuer l'adaptation de façon directe au travers les webservice pour dispositif maintenant associé au *container*, soit on décrit un aspect d'assemblage qui effectue ces modifications de manière automatique. On peut donc réaliser un contrat sur un composant en utilisant les aspects d'assemblage.
Nous allons *réutiliser* plusieurs composants pour constituer l'application finale intégrant le contrat.
Nous utilisons dans les exemples traités quatre composants de base :
– If, composant conditionnel
– Translator, composant qui doit respecter la QoS
– ContratTimeout, composant élémentaire spécifique au domaine des contrats
– RFID, composant générateur d'événements périodiques
Certains correspondent aux opérateurs d'ISL4WComp (ex. *If*), d'autres sont fournis par l'application (ex. Proxy du web service *Translator*, Proxy du web service pour dispositif *RFID*) et d'autres enfin correspondent à des composants de base nécessaires à la mise en œuvre de contrats (ex. ContractTimeout pour la vérification du timeout).
Le code des composants a priori définis dans la plateforme sont décrits ci-dessous.

Le composant If reçoit un événement sur son port fourni Input. Ensuite, il sauvegarde les paramètres de cet appel et émet l'événement RCheck. La réponse du check sera attendue sur le port fourni Check, sous forme de booléen. Un événement Then sera émis si le booléen vaut vrai, avec comme valeur le paramètre sauvegardé lors de l'appel à Input, ou un événement Else sera émit s'il vaut faux, avec pour valeur la valeur de la propriété ElseMessage.

```
[Bean]
public class If
{
        //------------------ Declarations --------------------
        public delegate void StringEventHandler(string s);
        //--------------------Interface ----------------------
        private string s_then, else_message;

        public string ElseMessage {                      // property
                get { return else_message; }
                set { else_message = value; }
        }
        public void Input(string s){                     // provided
                s_then = s;
                FireCheck();
        }
        public void Check(bool b){                       // provided
                if (b) FireThen(s_then);
                else FireElse(else_message);
        }
        public event StringEventHandler Then;            // required
        public event StringEventHandler Else;            // required
        public event EventHandler RCheck;                // required

        //----------------------Utilities --------------------
        private void FireThen(string s){
                if (Then != null) Then(s);
        }
        private void FireElse(string s){
```

```
                if (Else != null) Else(s);
        }
        private void FireCheck(){
                if (RCheck != null) RCheck(this, new EventArgs());
        }
}
```

Le composant Translator est un proxy de web service simple qui traduit une chaîne de caractères prise en paramètre de la méthode Translate et envoie le résultat dans l'événement Translated.

```
[Bean]
public class Translator
{
        //------------------ Declarations --------------------
        public delegate void StringEventHandler(string s);
        //--------------------Interface ----------------------
        public void Translate(string s){ ... }          // provided
        public event StringEventHandler Translated;     // required
}
```

Le composant proxy de web service pour dispositif RFID envoie régulièrement l'Id du TAG RFID à proximité. Avec la mise ne place d'un contrat on veut vérifier que cette périodicité est bien toujours respectée.

```
[Bean]
public class RFID
{
        //------------------ Declarations --------------------
        public delegate void StringEventHandler(string s);
        //--------------------Interface ----------------------
        public event StringEventHandler Id;                // required
}
```

Le composant ContractTimeout démarre un timer sur l'appel de méthode Start et le stoppe sur l'appel de la méthode Check. Ce dernier appel aura pour effet d'envoyer un événement Check si il survient avant l'expiration du timer. Si le timer expire avant cet appel, un événement Error est émis.

```
[Bean]
public class ContratTimeout
{
        //------------------ Declarations --------------------
        public delegate void BoolEventHandler(bool val);
        private Timer t;
        private bool hasExpired;
        object[] last_o;

        public ContratTimeout(){
                last_o = null;
                hasExpired = true;
                t = new Timer(100);
                t.Elapsed += FireError;
        }
        //--------------------Interface ----------------------
        public double Timeout {                          // property
                get { return t.Interval; }
```

```
                set { t.Interval = value; }
        }
        public void Start(param object[] o) {          // provided
                last_o = o;
                if (!t.isStarted()) {
                        hasExpired = false;
                        t.Start();
                }
        }

        public void Check(param object[] o) {          // provided
                if (last_o == o); // do nothing
                else
                   if (!hasExpired) {
                        t.Stop();
                        FireBoolEvent(true);
                   }
        }

        public event BoolEventHandler RCheck;           // required
        public event EventHandler Error;               // required

        //----------------------Utilities --------------------
        private void FireBoolEvent(bool b){
                if (RCheck!= null) RCheck(b);
        }
        private void FireError(){
                hasExpired = true;
                if (Error!= null) Error(this, new EventArgs());;
        }
}
```

Nous allons détailler maintenant deux exemples de mise en œuvre de contrats reposant sur la seule variation des aspects d'assemblage à partir de l'ensemble des composants présentés.

3.4.1 Premier exemple : contrat de timeout sur un WebService

Dans ce premier exemple, nous proposons d'illustrer la vérification du contrat de timeout sur un composant proxy de WebService.
Voici les points de coupe :

```
dummy = translator1
contract = contrat_timeout
```

Voici le greffon :

```
schema SA1 (dummy, contract) {
  dummy.^Translated     {  if ( contract.Check ) call  }
  dummy.Translate       {  contract.Start || call      }
}
```

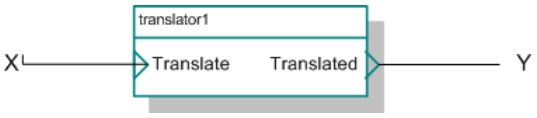

FIG. 3.6: Exemple 1 avant mise en œuvre du contrat de qualité de service

Nous avons connecté une textbox en entrée pour obtenir la phrase à traduire, et une autre en sortie sur l'évènement Translated pour afficher la traduction.

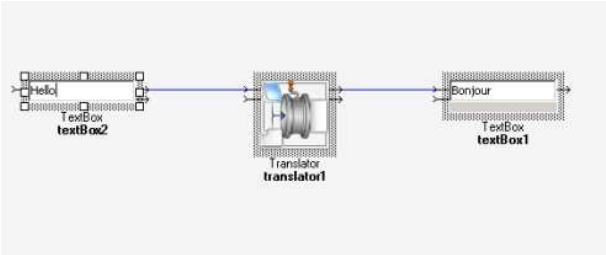

FIG. 3.7: Une application de l'exemple 1 dans le designer graphique avant mise en œuvre du contrat de qualité de service

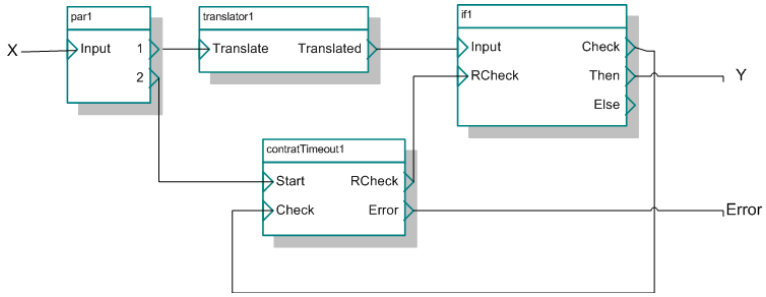

FIG. 3.8: Exemple 1 après mise en œuvre du contrat de qualité de service

L'ajout du contrat à l'assemblage se traduit donc par l'ajout des deux composants : *If* et *ContratTimeout*, plus ceux nécessaires pour traiter la violation de contrat. Ici, la violation déclenche l'envoi d'un message d'erreur qui sera affiché dans une textbox.

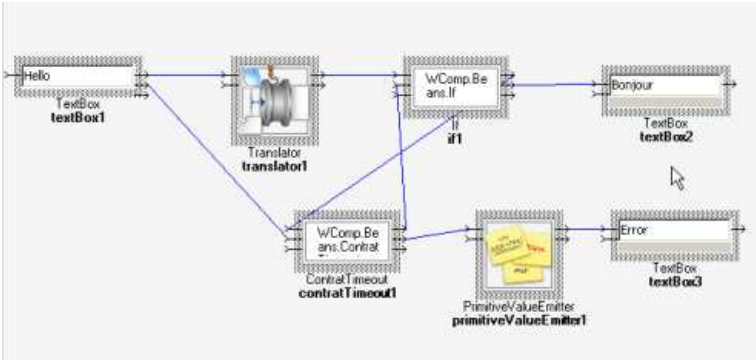

FIG. 3.9: Modificiation de l'application de l'exemple 1 dans le designer graphique après mise en œuvre du contrat de qualité de service

3.4.2 Second exemple : contrat sur un WebService pour dispositif

Dans ce troisième exemple, nous proposons d'étudier la vérification du contrat de timeout sur un composant proxy de WebService pour un Dispositif UPnP entre 2 événements consécutifs. Voici les points de coupe :

```
emitter = rFId1
contract = contract_timeout
```

Voici le greffon :

```
Schema SA2 (emitter, contract) {
  emitter.^Id     {  if ( contract.Check ) call  }
  contract.Check  { call ; contrat.Start }
}
```

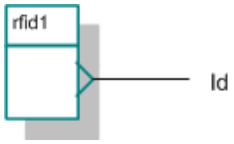

FIG. 3.10: Exemple 2 avant mise en œuvre du contrat de qualité de service

Dans ce cas simple, le composant proxy du web service pour dispositif d'identification RFId émet un événement représentant une personne reconnue par exemple. Cet événement est affiché dans une textbox.

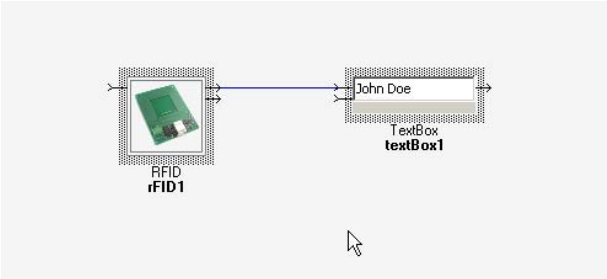

FIG. 3.11: Application de l'exemple 2 dans le designer visuel avant mise en œuvre du contrat de qualité de service

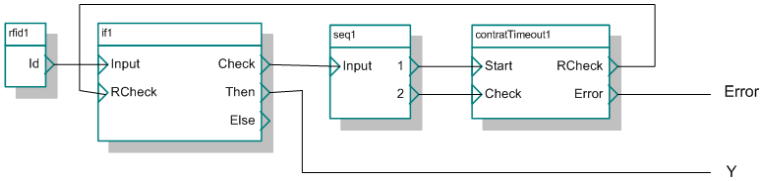

FIG. 3.12: Exemple 2 après mise en œuvre du contrat de qualité de service

Nous utilisons ici les mêmes composants pour vérifier l'intervalle de temps entre deux événements, que pour vérifier le temps d'exécution d'un web service, comme nous l'avons vu auparavant. Alors que le flot de contrôle initial démarrait le timer du contrat, c'est maintenant l'événement sortant du composant proxy du web service pour dispositif qui le fait. L'assemblage resultant est plus lisible, mais pourtant le nombre de composants et de liaisons ajoutés est le même. Nous retrouvons ici aussi la sortie d'erreur affichant un message dans une textbox.

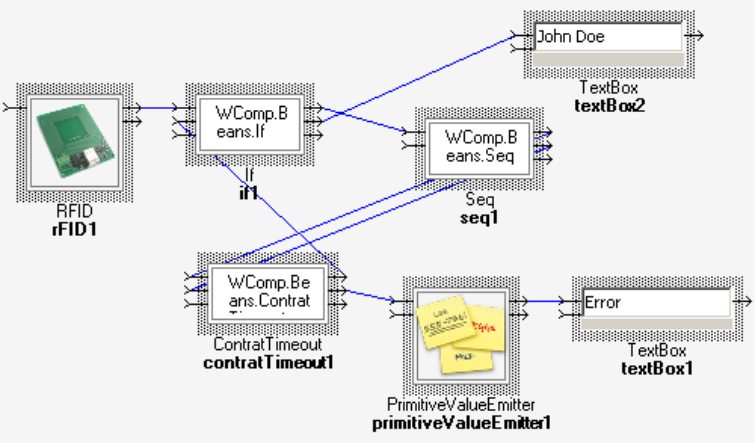

FIG. 3.13: Modification de l'application de l'exemple 2 dans le designer visuel après mise en œuvre du contrat de qualité de service

3.4.3 Cycle de vie des contrats

Dans WComp les contrats sont vérifiés à l'exécution. Les contrats peuvent être mis à jour par modification directe de l'assemblage de composants de l'application ou par tissage d'aspects d'assemblage. Dans le premier cas, les contrats sont mis à jour par ajout/retrait d'un composant spécifique, généré et inséré dans l'assemblage. Nous n'avons pas choisi de présenter ici cette approche. Dans le second cas, nous nous basons sur un ensemble de composants réutilisables tel que *ContratTimeout* de notre exemple pour la vérification de divers contrats. Les aspects d'assemblage permettent alors une description, dans le principe, sans génération de composant a posteriori (figure 3.14). Les contrats sont ainsi mis à jour par ajout/retrait et tissage d'aspects d'assemblage. Une généralisation de cette approche nécessiterait néanmoins la définition d'un ensemble de composants de base qui ne limiterait pas l'expressivité des contrats mis en œuvre.

3.5 Conclusion

Ce chapitre a présenté la plate-forme WComp et ses adaptations pour FAROS. WComp est à l'origine une plate-forme conçue pour concevoir des applications multi-dispositifs en Informatique Ambiante. Elle s'appuie sur une infrastructure à la fois composée de Web Services standards mais aussi de Web Services pour Dispositifs (UPnP) dotés de mécanismes de *recherche et découverte* et de *communication par événement* spécifiques. WComp est une plate-forme à composants logiciels construite autour d'un modèle de communication par diffusion d'événement. Ce modèle a été choisi car plus adapté à la programmation d'applications mettant en œuvre des entrées/sorties telles que celles des Web Services pour dispositifs dans un environnement réel.

La plate-forme WComp pour FAROS permet l'orchestration de web services et web services pour dispositifs. Elle permet de concevoir ainsi par réassemblage de composants des Web Services pour Dispositifs adaptés et de plus haut niveau. Enfin WComp permet une modification des assemblages de composants au travers divers outils à vocations différentes appelés *designers*. Nous avons décrit dans ce chapitre l'utilisation possible d'un *designer* par Aspects d'Assemblage pour la mise en œuvre et la vérification de contrat dans une application WComp.

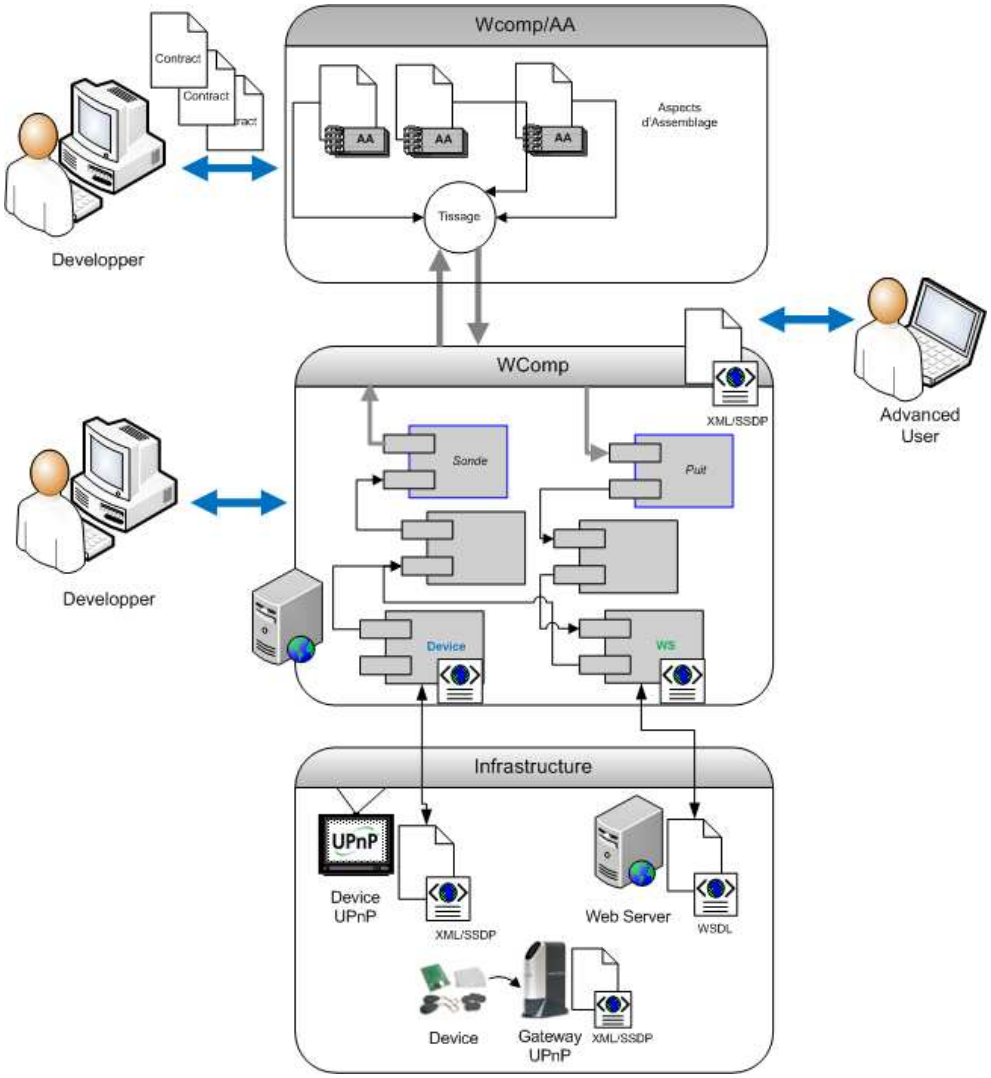

FIG. 3.14: Architecture général de la plate-forme WComp FAROS

Bibliographie

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, July 1999.

[CFWBFT+06] Daniel Cheung-Foo-Wo, Mireille Blay-Fornarino, Jean-Yves Tigli, Anne-Marie Pinna-Déry, David Emsellem, and Michel Riveill. Langage d'aspects pour la composition dynamique de composants embarqués. *L'objet : coopération dans les systèmes à objets*, 12(2-3) :89–112, April 2006.

[CFWTLR06] Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stéphane Lavirotte, and Michel Riveill. Wcomp : a Multi-Design Approach for Prototyping Applications using Heterogeneous Resources. In *17th IEEE International Workshop on Rapid System Prototyping (RSP)*, Chania, Crete, June 2006.

[CFWTLR07] Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stéphane Lavirotte, and Michel Riveill. Contextual Adaptation for Ubiquitous Computing Systems using Components and Aspect of Assembly. In *Applied Computing (IADIS)*, Salamanca, Spain, February 2007. IADIS.

[HLT06] Vincent Hourdin, Stéphane Lavirotte, and Jean-Yves Tigli. Comparaison des systèmes de services pour dispositifs. Technical Report I3S/RR-2006-25-FR, Laboratoire I3S, Sophia Antipolis, France, August 2006.

[HLT07] Vincent Hourdin, Stéphane Lavirotte, and Jean-Yves Tigli. Service UPnP pour dispositifs autonomes. In *Service UPnP pour dispositifs autonomes*, volume H5002. Techniques de l'Ingénieur, February 2007.

[Sei06] Lionel Seinturier. *Développement de logiciels par aspects : JFDLPA 2005*, volume 12. Hermès, lavoisier edition, Juin 2006.

[TCFWLR06] Jean-Yves Tigli, Daniel Cheung-Foo-Wo, Stéphane Lavirotte, and Michel Riveill. Adaptation au contexte par tissage d'aspects d'assemblage de composants déclenchés par des conditions contextuelles. *RTSI Série ISI*, 11(5) :89–114, 2006.

[WS-a] Web Service Dynamic Discovery (WS-Discovery).

[WS-b] Web Services Eventing (WS-Eventing).