La programmation orientée service vue de l'utilisateur final

Nassim Laga

Ecole Doctorale EDITE

Thèse présentée pour l'obtention du diplôme de Docteur de Télécom & Management SudParis

Doctorat conjoint TMSP-UPMC

Spécialité : Informatique et Télécommunications

Par Nassim LAGA

Service-Oriented Computing from the User Perspective

Soutenue le **17/11/2010 devant le jury composé de :**

Tiziana MARGARIA	**Professeur à l'université de Potsdam**	Rapporteur
Roch H. GLITHO	**Professeur à l'université de Concordia**	Rapporteur
Guy PUJOLLE	**Professeur à Paris 6**	Examinateur
Stéphane FRENOT	**Professeur à l'INSA Lyon**	Examinateur
Yvon KERMARREC	**Professeur à Télécom Bretagne**	Examinateur
Emmanuel BERTIN	**Docteur à Orange Labs**	Encadrant
Noel CRESPI	**Professeur à Télécom SudParis**	Directeur de thèse

Thèse n°

2010TELE0024

Acknowledgments

Mes premiers remerciements vont, bien entendu, à mes encadrants : le docteur Emmanuel Bertin et le professeur Noel Crespi. Le sujet de thèse proposé est à la fois passionnant sur le plan théorique et précurseur d'applications industrielles, ce qui m'a offert durant ces trois années, deux axes de montée en compétence. Je les remercie aussi de leurs encouragements incessants, de leur aide et de la veille qu'ils ont mise en œuvre pour que cette thèse soit une réussite.

Je remercie chaleureusement le professeur Guy Pujolle, le professeur Tiziana Margaria, le professeur Roch Glitho, le professeur Yvon Kermarrec et le professeur Stéphane Frenot d'avoir accepté de participer à mon jury de thèse.

Je remercie très particulièrement Orange Labs de m'avoir donné l'opportunité de réaliser ce travail, ainsi que tous les moyens nécessaires pour son bon déroulement. Je remercie Philippe Michon et Olivier Bouillon de m'avoir fait confiance et de m'avoir permis de rejoindre le groupe. Je remercie également Pascal Lesieur et Marc Mazoué, respectivement responsable d'unité à Orange Labs, de m'avoir accueilli dans leurs équipes durant la thèse. Mes remerciements vont aussi à Frédéric Delmond et Guillaume Gautier, responsables de laboratoire, qui, en plus de m'avoir donné tous les moyens nécessaires pour la réalisation de ce travail, m'ont permis de continuer cette aventure passionnante avec Orange Labs. Enfin, je remercie Cécile Maillot et tous les membres de ma nouvelle équipe pour leur sympathie et leur accueil dans mes nouvelles fonctions.

Ma gratitude va aussi à tous les gens qui ont participé de près ou de loin à la réalisation de ce travail, que ce soit de façon informelle dans les couloirs et les pauses café, ou formelle à travers notamment les projets CCKMA, Framework de présentation, et SERVERY. Je remercie plus particulièrement Jean-pierre Deschrevel, Julien Van Den Bossche, Maryline Gidon, Ivan Bedini, Zhenzhen Zhao, Abderahmane Maaradji, Cuiting Huang, Sivasothy Shanmugalingam, Khalil Leghari et Mariano Belaunde pour les nombreux échanges fructueux que nous avons eus. Je remercie également les membres de l'équipe RS2M de l'Institut Telecom Sudparis, et des unités JTE, ASC, et NCIS d'Orange Labs. Leurs commentaires, suggestions, ou tout simplement sympathie ont largement contribué au succès de cette thèse.

Merci enfin à mes proches : mes parents, mes frères et sœurs, ainsi que tous mes amis. Cette thèse leur est dédiée.

Bibliography

1 International Journals

- Laga, N., Bertin, E., Crespi, N. Bringing runtime service composition to Web desktop environments: implementation, feedback and lessons learned. To be submitted to International Journal of Web Services Research.
- Laga, N., Bertin, E., Bedini, I., Crespi, N., Molina, B., Zhao, Z. User-centric service selection: the concept of abstract Widget. Submitted to Springer World Wide Web Journal.
- Sethom, K., Ali-Yahiya, T., Laga, N., Pujolle, G. 2008. A QoS-aware mesh protocol for future home networks using autonomic architecture. EURASIP Journal on Wireless Communications and Networking. pp.1-9. January, 2008.

2 International Conferences and Workshops

- Laga, N., Bertin, E., Crespi, N. 2010. Promoting Mashup Creation through Unstructured Data Extraction. To be submitted to the 14th International Conference Business Information Systems.
- Laga, N., Bertin, E., Crespi, N. 2010. Composition at the Frontend: the User Centric Approach. Accepted in ICIN 2010. Berlin, Germany. October 2010.
- Laga, N., Bertin, E., Crespi, N. 2010. Business Process Personalization through Web Widgets. In proceedings of IEEE International Conference on Web Services. ICWS 2010. pp. 551-558. Miami, USA. July 2010.
- Laga, N., Bertin, E., Crespi, N. 2010. Widgets to facilitate service integration in a pervasive environment. In proceedings of IEEE International Conference on Communication. ICC 2010. vol., no., pp.1-5. Cape Town, South Africa. May 2010.
- Laga, N., Bertin, E., Crespi, N. 2009. Building a user friendly service dashboard: Automatic and non-intrusive chaining between widgets. In proceedings of the 2009 Congress on Services – I. SERVICES. IEEE Computer Society, Washington, DC, 484-491. Los Angeles, USA. July 2009.
- Laga, N., Bertin, E., Crespi, N. 2009. A web based framework for rapid integration of Enterprise applications. In proceedings of ACM International Conference on Pervasive Services. ICPS 2009. ACM, New York, NY, 189-198. London, United Kingdom. July 2009.

- Zhao, Z., Laga, N., Crespi, N. 2009. A Survey Of User Generated Service. In proceedings of IEEE International Conference on Network Infrastructure and Digital Systems. IC-NIDC 2009. vol., no., pp.241-246. Beijing, China. Nov. 2009.
- Zhao, Z., Laga, N., Crespi, N. 2009. The Incoming Trends of End-user driven Service Creation. In proceedings of International Conference on Digital Businesses. DigiBiz 2009. LNICST 21, pp. 98–108, 2009, Springer-Verlag. London, United Kingdom. June 2009.
- Laga, N., Bertin, E., Crespi, N. 2008. A unique interface for web and telecom services: From feeds aggregator to services aggregator. In ICIN 2008. Bordeaux, France. October 2008.
- Laga, N., Bertin, E., Crespi, N. 2008. User-centric Services and Service Composition, a Survey. In proceedings of the 32nd Annual IEEE Software Engineering Workshop. SEW. IEEE Computer Society, Washington, DC, 3-9. Kassandra, Greece. October, 2008.

3 Book Chapters

- Sethom, K., Laga, N., Pujolle, G. 2008. QoS Management in Autonomic Home Networks. Home Networking, Springer IFIP International Federation for Information Processing Series. Paris, France. Vol.256. pp.101-110.

4 Patents

- Bedini, I., Bertin, E., Laga, N. 2009. Procédé d'exécution d'un Service Applicatif Dans Un Environnement Web. N° INPI: 0954427.
- Bertin, E., Laga, N., Van den bosh, J. Procédé De Communication Entre Applications Exécutées Dans Des Navigateurs Distincts. N° INPI: 0956654
- Bertin, E., Laga, N., and Deschrevel, J.P. 2008. Procédé Et Système De Communication Entre Applications Web Distinctes. N° INPI: 0856592.

Abstract

The last decade has attracted lot of research work in Service-Oriented Computing (SOC), giving raise to standardized architectures, protocols, and technologies that enable developers to easily expose and reuse services. However, these technologies do not fully consider the users as potential actors in the creation of services based on existing ones, as advocated in Web 2.0 paradigm. In this thesis, after a deep investigation of SOC and its intrinsic SOA paradigm, we propose a new approach based on Widgets. We propose the Widget-Oriented Architecture (WOA); a new paradigm to enable a user-centric service reuse. In addition, we introduce new innovative mechanisms based on the WOA paradigm to overcome current limitations of SOA in service composition and business process management fields. This new paradigm, along with the innovative architecture and mechanisms introduced, has been validated through implementation and testing.

Résumé

SOC, pour *Service-Oriented Computing*, est un paradigme d'ingénierie qui a attiré beaucoup de travaux de recherche ces dernières années. Ces travaux ont donné lieu à des architectures, protocoles, et technologies standards, afin de permettre à des développeurs d'exposer des services et en réutiliser d'autre publiés par des tiers. Cependant, ces technologies sont actuellement limitées aux besoins des développeurs uniquement. L'utilisateur final n'est malheureusement pas considéré comme un acteur potentiel dans le processus de réutilisation de services. Ainsi, contrairement aux principes Web 2.0 qui mettent l'utilisateur final au cœur du processus de génération de contenus et de services, les technologies actuelles de SOC se focalisent plus sur les développeurs. Dans cette thèse, après une étude approfondie de SOC et son paradigme intrinsèque (SOA pour *Service-Oriented Architecture*), nous proposons un nouveau paradigme basé sur le concept de Widget : WOA (pour *Widget-Oriented Architecture*), un nouveau paradigme qui vise à permettre la réutilisation de service centrée sur les besoins de chaque utilisateur (*user-centric*). Basé sur ce nouveau paradigme, nous introduisons de nouveaux mécanismes qui répondent aux limitations des architectures SOA dans les domaines de la composition de services et de la gestion de processus métiers (BPM pour *Business Process Management*). Ce travail est validé à travers une implémentation et plusieurs démonstrations/expérimentations.

Table of Contents

Figures

Tables

French Summary

Introduction

1 Contexte

Introduit par Tim O'Reilly, le Web 2.0 désigne un ensemble de principes qui caractérisent l'évolution des pratiques, des usages, et des technologies du Web de l'ère initiale, dite statique, à l'ère actuelle. Les principes Web 2.0 peuvent se résumer par six points :

- création de services réutilisables au lieu d'applications monolithiques ;
- la participation de l'utilisateur dans la création de contenus et de services ;
- la conception orientée utilisateur final ;
- le partage d'information ;
- l'interopérabilité ;
- et les interfaces riches.

Ces principes ont révolutionné l'ingénierie logicielle ainsi que la façon dont l'utilisateur interagit avec l'ensemble des applications qu'il utilise. Les logiciels ne sont plus en effet packagés en applications monolithiques. Ils sont fragmentés en un ensemble de services (*Web services*) qui sont ensuite publiés sur et réutilisé à travers le Web. Cette approche permet de promouvoir d'une part le partage et la réutilisation de services à travers le réseau Internet, et la collaboration et l'intégration de service inter-organisationnel d'autre part. Quand aux utilisateurs, ils n'utilisent plus les applications de la même manière. Les applications Web, qui sont disponibles sur le réseau, sont de plus en plus matures et sophistiquées, et tendent à remplacer les applications traditionnelles que les utilisateurs installent sur leurs machines. De plus, les applications Web 2.0 donnent un rôle très actif à l'utilisateur, en lui offrant des outils de personnalisation et en lui permettant de générer lui-même du contenu. Les exemples typiques de ce phénomène sont les sites Web Wikipédia[1] et Youtube[2], dont succès est complètement dépendant de la qualité et la quantité du contenu généré par les utilisateurs.

Ces deux observations (la fragmentation du Web en services réutilisables, et le rôle actif de l'utilisateur dans l'écosystème des services), ont donné lieu à de nombreux travaux de recherches, qui

[1] Wikipedia, http://en.wikipedia.org/wiki/Main_Page, visité le 10 sept. 2010
[2] YouTube, http://www.youtube.com/, visité le 10 sept. 2010

visent à tirer profit du rôle actif des utilisateurs finaux en leur permettant de créer eux même des services combinant ceux qui existent déjà. Les deux approches émergentes pour répondre à ce besoin sont la composition automatique, et la composition semi-automatique. La composition automatique permet à l'utilisateur de créer des services en exprimant son besoin en langage naturel, et la composition semi-automatique permet à l'utilisateur de créer un service composé en enchainant graphiquement des services.

Par ailleurs, les environnements de services ont évolué en parallèle avec l'évolution du Web. Nous appelons un environnement de services tout logiciel permettant à l'utilisateur de consommer des services. Ceci inclut les systèmes d'exploitation traditionnels (e.g. Windows et Linux), ainsi que les systèmes d'exploitation Web (e.g. Google Chrome OS et eyeOS[3]), tant comme les agrégateurs de services (e.g. iGoogle[4] et Netvibes[5]). L'évolution vers les agrégateurs de services présente une analogie intéressante avec l'évolution du Web 1.0 vers le Web 2.0, aussi bien que du point de vue technique que conceptuel. Techniquement, les agrégateurs de services se basent sur le concept de Widget ; une interface utilisateur qui permet de consommer un service unitaire accessible sur le Web. Conceptuellement, les agrégateurs de services sont des applications Web 2.0 dans la mesure où ils permettent à l'utilisateur de personnaliser son environnement en chargeant uniquement les services dont il a besoin.

Cependant, contrairement aux services Web (*Web services*) et aux architecture SOA, sur lesquelles de nombreux travaux de recherche ont été conduits, le domaine des Widgets et des agrégateurs de Widgets est moins étudié. Les potentialités de composition de Widget ne sont n'est en effet pas approfondies. Cette thèse vise à répondre à cette limitation.

2 Problématiques

SOA (pour *Service-Oriented Architecture*) est un paradigme d'ingénierie, qui est caractérisé par la réutilisation de services à travers un registre de service commun et une API de publication commune à tous les fournisseurs de service. Ceci est l'aboutissement de plusieurs années de recherche dans le domaine. WSA (pour *Web Service Architecture*) [Newcomer, 2002] et REST (pour *Representational State Transfer*) [Fielding, 2000] sont sans doute les technologies les plus utilisées actuellement pour réaliser une architecture SOA. Elles sont appliquées dans différents domaines tel que la composition de services et la gestion des processus métiers. Cependant, SOA a de nombreuses limites lorsqu'on considère les principes du Web 2.0, qui font de l'utilisateur un acteur majeur dans le processus de création de services. Plus précisément :

---

[3] eyeOS, http://www.eyeos.com/, accessed Dec 22, 2009
[4] iGoogle, http://www.google.com/ig?hl=en, accessed Dec 22, 2009
[5] Netvibes, http://www.netvibes.com, accessed Dec 22, 2009

- SOA n'est pas orienté utilisateur. Il est en effet conçu pour répondre aux besoins de réutilisation des développeurs. Par conséquence, les outils de composition de service actuel ne sont pas accessibles par l'utilisateur final.
- SOA ne traite pas l'aspect interface utilisateur des services.
- SOA ne permet pas la composition de service à base de données non structurées. Ceci est particulièrement utile dans les services de communication, où des données significatives sont échangées entre les utilisateurs (e.g. les numéros de téléphone, des adresses postales…etc.). Ces données ne sont pas aujourd'hui considérées dans le paradigme SOA.
- SOA ne considère pas la prolifération des terminaux utilisateurs. Il n'est en effet pas possible aujourd'hui à l'utilisateur final de composer deux services chargés sur deux terminaux différents. Ceci permettrait de composer par exemple un service de mail chargé sur un mobile avec un service de lecture vidéo chargé sur la télé, afin de lire une vidéo en pièce jointe.
- A travers la description et la publication d'interface, SOA diminue significativement le couplage entre les intégrateurs de services et les fournisseurs de services qu'ils utilisent. Cependant, la suppression du service, ou la modification de l'interface, entrainera forcément un disfonctionnement dans tous les services qui l'utilisent.

3 Contributions de la thèse

A travers l'état de l'art nous montrons que le paradigme SOA répond uniquement aux besoins des développeurs. Ceci est une importante limitation lorsqu'on considère les principes du Web 2.0 qui font de l'utilisateur un acteur majeur dans le cycle de vie des services. D'autre part, nous montrons aussi que les mécanismes actuels des Widgets et des agrégateurs de Widgets sont orientés utilisateur, mais ne prennent malheureusement pas en compte la dimension création et composition de services. Notre contribution dans cette thèse est la définition d'un nouveau paradigme qui tire le meilleur des deux domaines afin de satisfaire les développeurs et les utilisateurs finaux. Nous introduisons ainsi WOA (pour *Widget-Oriented Architecture*), que nous appliquons à deux domaines où SOA est largement utilisé aujourd'hui : la composition de services, et la gestion des processus métiers. En se basant sur ce nouveau paradigme, nous définissons d'une part un ensemble d'outils innovants de composition de services destinés aux utilisateurs finaux, et nous introduisons d'autre part une nouvelle méthode de gestion et d'automatisation des processus métiers qui vise à mieux gérer l'hétérogénéité et l'instabilité des processus métiers.

Etat de l'art

Cette thèse a comme objectif de définir une alternative à SOA qui sera plus orienté vers l'utilisateur (*user-centric*). Dans ce chapitre nous allons donc étudier les différentes technologies SOA d'une part, puis les agrégateurs de Services d'autre part.

1 SOA (Service-Oriented Architecture)

SOA est le paradigme qui fournit des facilités aux développeurs afin de promouvoir la réutilisation de services. La figure 1 montre les différents rôles ainsi que leurs interactions.

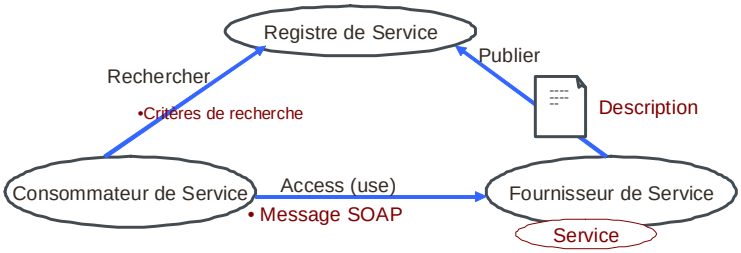


Figure 1: SOA.

SOA se base sur un certain nombre de principes, que chaque rôle doit accomplir. Nous les résumons comme suit :

- Le registre de services doit fournir des interfaces de publication et de découverte de services.
- Les développeurs doivent créer des services et publier leurs descriptions dans le registre de services.
- Les développeurs peuvent réutiliser les services existants dans la création de leurs propres services.

Plusieurs technologies permettent la réalisation de ce pattern (e.g. CORBA, et EJB), mais WSA (pour *Web Services Architecture*) est sans doute la technologie la plus utilisée actuellement. Un service Web [W3C, 2004b] (ou *Web Service* en anglais) est *un système conçu pour facilité les interactions entre machines à travers un réseau. Il a une interface décrite par un format interprétable par les machines (WSDL). Les autres systèmes interagissent avec un Web service en utilisant des messages SOAP d'une manière prescrite par la description. Les messages SOAP sont typiquement transmis sur HTTP avec une sérialisation XML*. Cette définition résume les différentes technologies utilisées dans WSA : WSDL, SOAP, HTTP, et SOAP. La description d'un Web service est souvent publiée dans un registre de services, typiquement UDDI (pour *Universal Description, Discovery and Integration*).

WSDL-SOAP offre beaucoup d'avantages aux développeurs : réutilisation, composition, et couplage faible. Cependant, ces deux technologies ne sont pas accessibles par l'utilisateur final. Il ne peut pas en effet pas réutiliser les services existants pour la création d'un nouveau qui correspondrait à ses propres besoins.

Pour répondre à ce problème, les approches émergentes tentent de rajouter de la sémantique dans la description de service afin de pouvoir de composer les services automatiquement, à partir d'une expression des besoins de l'utilisateur. Les technologies sémantiques telles-que RDF (pour *Resource Description Language*), OWL (pour *Web Ontology Language*), et SA-WSDL (pour *Semantic Annotation WSDL*) sont alors utilisées.

1.1 La composition de service basée sur SOA

Nous classifions les techniques de composition de services en trois catégories : la composition statique, la composition semi-automatiques, et la composition automatique. Dans cette section nous allons détailler chacune de ces catégories.

a. *La composition statique*

Nous appelons composition statique les mécanismes permettant à des développeurs, mais pas aux utilisateurs, de créer des services à partir de ceux qui existent déjà. Le terme statique indique ainsi le fait que l'utilisateur final n'a pas la main sur les services composés. Il ne peut ni créer de nouveaux services composés ni modifier ceux qui existent déjà selon ses propres besoins. Ce type de composition est souvent réalisé à travers des APIs de programmation conçues pour le développeur (e.g. *Java SOAP client*, *PHP SOAP*, *IBM Dojo toolkit extension*, et *jQuery SOAP client*).

b. *La composition automatique*

A l'opposition de la composition statique, la composition automatique est destinée à l'utilisateur final. Elle se base sur des outils de composition qui permettent à l'utilisateur final de créer des services composés à partir d'une expression de besoins en langage naturel. Son pattern est illustré dans la Figure 2.

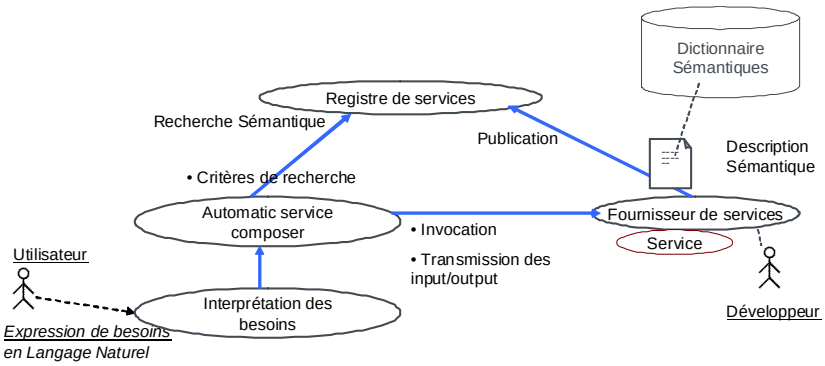


Figure 2 : Composition automatique.

c. La composition semi-automatique

La composition semi-automatique vise actuellement une communauté d'utilisateurs qui ont un profil entre développeur et utilisateur final. Elle offre des outils graphiques qui leur permettent de définir des enchainements de services sans avoir des connaissances approfondies dans le développement de logiciels. Cependant, des connaissances de base sur ce qui est un Web services, les inputs, les outputs, ainsi que le concept d'organigramme informatique (*flowchart*) sont nécessaires.

Ce type de composition est essentiellement poussé par le succès des sites Web 2.0, où l'utilisateur est souvent générateur de contenu ; l'idée étant de lui permettre d'être aussi un générateur de services. Dans cette catégorie, nous distinguons les outils de composition basés sur les environnements de bureau. Nous pensons que ce sont les plus proches de l'utilisateur final d'une part, et qu'ils nécessitent le moins de connaissances en termes de développement logiciel d'autre part. Ces outils permettent à l'utilisateur final de combiner des services directement à partir de son environnement de travail. Les exemples typiques de ce type de composition sont Microsoft OLE (pour *object linking and embedding*), le copier/coller, ou le glisser/déposer. Cependant, ces mécanismes, basés sur les systèmes d'exploitation traditionnels, sont premièrement limités par le fait qu'ils ne considèrent pas la composition des services Web d'une part, et du manque de mécanismes sémantique, qui permettraient d'anticiper certaines compositions, d'autre part.

1.2 La gestion des processus métiers basée sur SOA

La gestion des processus métiers consiste à chercher et à découvrir l'ensemble des processus d'une organisation, les modéliser, les automatiser, et les faire évoluer en fonction de nouveaux besoins. Dans cette section nous allons résumer les pratiques courantes pour réaliser ces actions.

a. Découverte et modélisation des processus métiers

La tache de découverte et de modélisation des processus métiers est actuellement réalisée soit en suivant une méthode descendante (*top-down*) ou une méthode ascendante (*bottom-up*). La méthode descendante consiste à modéliser une vue globale d'un processus métiers, puis le décliner en plusieurs versions en modélisant les détails spécifiques à chaque entité, rôle, ou personne. La méthode ascendante, quand à elle, commence par capturer les habitudes (processus) de chaque entité, rôle, ou personne, puis essayer de les généraliser afin d'avoir un nombre limité de processus finaux.

Dans les deux méthodes, le résultat final est un compromis entre le niveau de détails automatisé et la minimisation du nombre de processus. C'est donc un ensemble processus métiers qui est réduit mais qui automatise les taches les plus fréquentes des utilisateurs. Les détails spécifiques à chaque utilisateur ne sont malheureusement souvent pas automatisés. En plus d'être hétérogènes et restreints à des ensembles réduits d'utilisateurs, ces détails sont souvent très dynamiques.

b. Automatisation des processus métiers

La problématique essentielle qui doit être considérée par les applications de gestion de processus métiers est de répondre rapidement aux changements et aux évolutions. Les approches existantes tentent d'accélérer le plus possible le temps de développement des applications pour automatiser les processus métiers. WSA est sans doute la technologie la plus utilisé dans ce domaine (associée ou non à des outils de composition de services comme BPEL). Les différentes tâches des processus métiers sont implémentées et exposées en tant que services Web, puis assemblées suivant la logique d'un processus afin de l'automatiser.

Les limitations qu'on peut constater à travers ces deux points peuvent se résumer en :

- Le temps d'automatisation d'un processus métier reste long du moment que deux actions, réalisées par deux entités différentes sont nécessaires : la découverte et la modélisation des processus, puis le développement de l'application qui les automatise.
- Les détails sont difficilement automatisables.
- L'adaptation à de nouveaux besoins est longue.
- Le couplage entre les implémentations est fort

1.3 Conclusions

Plusieurs technologies permettent aujourd'hui de réaliser une architecture de services. Ces technologies sont conçues essentiellement pour répondre aux besoins des développeurs en termes de réutilisation et d'accélération du temps de développement. L'utilisateur final n'est actuellement pas considéré. Par conséquence, la composition de service reste un domaine réservé aux développeurs. Ceci pose aussi de sérieuses limitations concernant la gestion des processus métiers. Le Tableau 1 récapitule les avantages et les limitations de SOA dans la composition de services et la gestion des processus métiers.

Tableau 1. Les avantages et les inconvénients de SOA.

		Avantages	**Incovénients**
Composition de Services	Composition statique	• Couplage faible en les services basiques. • Création d'applications distribuées. • Les services crées répondent aux besoins de l'utilisateur.	• Conçue seulement pour les développeurs. • Couplage fort entre les services composés et ceux qu'ils utilisent.
	Composition Semi-automatique	• Conçue pour les utilisateurs avancés (Pas nécessairement des développeurs). • Le TTM est faible lorsque l'utilisateur est un utilisateur avancé. • Les services crées respondent aux besoins de l'utilisateur.	• Les utilisateurs ordinaires ne peuvent pas créer des services. Par conséquent, le TTM reste important pour eux. • Les services créés ne sont pas riches • Impossible de créer des services distribués sur différent terminaux de l'utilisateur. • Couplage fort entre les services composés et ceux qu'ils utilisent.
	Composition automatique	• Conçue pour les utilisateurs ordinaires. • TTM faible.	• Les services crées ne répondent souvent pas exactement aux besoins de l'utilisateur. • Impossible de créer des services distribués sur différent terminaux de l'utilisateur. • Couplage fort entre les différents services en termes de sémantique. • Couplage fort entre les services composés et ceux qu'ils utilisent.
Gestion des processus métiers		• Réutilisation et implémentation rapide (par des développeurs) de processus métiers • Les outils graphiques tels que BPEL accélèrent considérablement le développement des processus métiers.	• Les détails de processus, qui sont souvent spécifiques à une population limitée, sont rarement automatisés. • L'adaptation à de nouveau processus est longue. • Couplage fort entre les développeurs des processus et les Web Service qu'ils utilisent. • Les données non structurés ne sont pas capturées.

2 Agrégateur de Services

SOA améliore considérablement les interactions dites *machine-to-machine*. Cependant, comme nous l'avons détaillé dans le point précédent, les interactions homme-machine ne sont pas prises en compte. Dans ce domaine nous mettons l'accent sur les environnements de travail (connus aussi sous le nom de *virtual desktops*) en général, et les agrégateurs de services en particulier (ou agrégateurs de Widgets). Ces environnements se focalisent sur l'aspect interaction avec l'utilisateur final. De plus, les agrégateurs de Widgets (e.g. Netvibes, et iGoogle), qui ont émergé avec le paradigme Web 2.0, permettent à l'utilisateur une personnalisation accrue. Les utilisateurs peuvent personnaliser fonctionnellement leur espace de travail en chargeant des fonctionnalités au lieu d'utiliser directement des applications packagés.

Les agrégateurs se basent sur le concept de Widget (connu aussi sous le nom de Gadget ou Portlet). Nous distinguons trois technologies réalisant ce concept : JSR 168/286 (pour *Java Specification Request*) [Sun, 2003], spécifications W3C [W3C, 2007], et UWA (pour *Universal Widget API*) [UWA, 2008]. JSR 168/286 est une spécification Java qui décrit une Portlet comme étant un composant Web, géré par un container, et qui traite des requêtes et génère du contenu dynamiquement. Les Portlets sont utilisées par des portails Web comme interfaces utilisateur qu'on peut rajouter ou supprimer dynamiquement. La spécification JSR 168/286 se focalise sur la standardisation des interactions entre la partie serveur d'une Portlet et le container. Le but est de permettre à la même Portlet d'être utilisée dans différents container de différents fournisseurs.

Une Widget (terminologie utilisée par W3C ainsi que dans UWA) est conceptuellement identique à une portlet. La différence se situe sur le plan technologique. Premièrement, elle n'est pas limité au langage de programmation Java ; la partie serveur d'une Widget peut être implémentée suivant n'importe quel langage (Php, Java, Python…etc). Cependant, la partie cliente est souvent limitée aux technologies interprétables par les navigateurs Web ((X)HTML, XML, JavaScript, Flash, Java Applets…etc). Deuxièmement, contrairement aux spécifications JSR 168/286 qui se focalisent sur les interactions entre la partie serveur d'une Portlet et le container, les Widgets se focalisent plus sur les interactions entre la partie client d'une Widget (UI) et l'agrégateur.

Les Widgets et Portlets présentent de sérieuses similitudes avec les services Web dans SOA. Conceptuellement, les deux technologies permettent d'exposer des fonctionnalités d'une application. Cependant, contrairement aux Widgets, les services Web on reçu beaucoup d'attention, ce qui a donnée lieu à de nombreuses technologies qui permettent leurs réutilisation et compositions. Les travaux sur la réutilisation et la composition de Widgets sont en effet rares. Nous distinguons néanmoins quelques articles (e.g. [Díaz, 2008], [Vo, 2006], [Sire, 2009], et [Soriano, 2006]) qui encouragent ce type de composition.

Les auteurs de [Soriano, 2006] ont meme proposé un agrégateur de Widget (EzWeb) et une API de développement de Widgets qui permet la composition de ces dernières lorsqu'elles sont chargées sur la même instance de l'agrégateur. Figure 3 montre comment cette composition est définie et comment elle est exécutée.

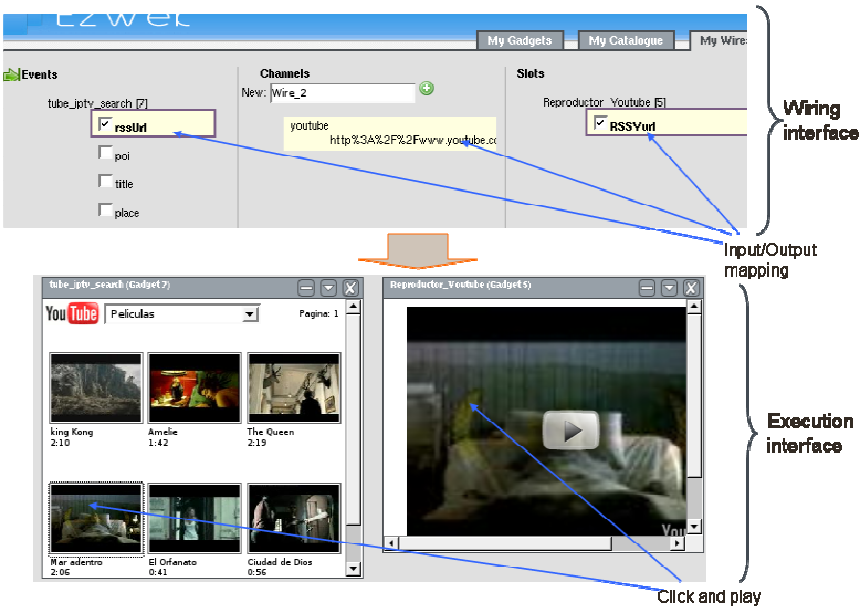


Figure 3 : La plateforme EzWeb.

3 Conclusion

Les conclusions les plus importantes qu'on peut tirer de cet état de l'art est le manque de prise en compte des interactions homme-machine des technologies SOA d'une part, et le manque d'outils de réutilisations et de compositions dans le domaine des agrégateurs de Widgets d'autre part. Les technologies SOA par leur nature vise en effet à répondre aux besoins des développeurs. Les interfaces graphiques, qui interagissent avec l'utilisateur final, sont en effet moins importantes. Les Widgets quand à elles présentent l'avantage d'être conçu essentiellement pour interagir avec l'utilisateur final. Par conséquent, l'aspect interface graphique prend toute son importance.

Cette étude des deux domaines (SOA et Widget) montre que chacun des deux a ses avantages et ses inconvénients, ce qui révèle un nouveau challenge ; celui de prendre le meilleur de chaque domaine afin de construire une architecture ou les développeurs comme les utilisateurs finaux puissent combiner des services.

Contributions

La contribution la plus importante de cette thèse est la définition d'un nouveau paradigme de programmation orientée service centré sur l'utilisateur final. Ce paradigme basé sur le concept de Widget est nommé WOA (pour *Widget-Oriented Architecture*). En s'appuyant sur ce nouveau paradigme, nous concevons un agrégateur de Widgets qui réalise l'ensemble des principes définis dans le paradigme. Nous déclinons ensuite ce paradigme ainsi que l'agrégateur conçu dans les domaines d'application de SOA : composition de services et la gestion des processus métiers.

1 WOA (Widget Oriented Paradigm)

Ce paradigme est basé sur le concept de Widget comme l'élément de base qui permet la réutilisation et la composition de services. Nous définissons une Widget comme une interface utilisateur qui donne accès à une implémentation du service offert. L'interface utilisateur est taguée sémantiquement afin de pouvoir réutiliser les capacités de la Widget dans d'autre Widgets.

Comme illustré dans la Figure 4, le paradigme WOA est caractérisé par cinq rôles : le développeur, le fournisseur, registre, le client, et l'utilisateur final. Le paradigme consiste en un ensemble de principes que chaque doit suivre. Les sous-sections suivantes résument ces principes.

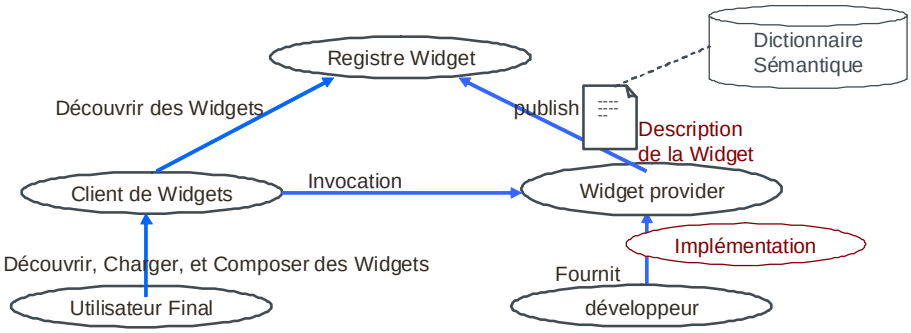


Figure 4 : Le paradigme WOA.

1.1 Les principes liés au registre de Widgets

Comme dans SOA, le registre de Widget de WOA doit fournir des interfaces de publications et de découverte de Widgets. De plus, il est recommandé de fournir un mécanisme de sélection de services parmi ceux qui sont fonctionnellement équivalents. Il important aussi que ce mécanisme soit paramétrable par l'utilisateur final. En d'autre termes, il est important que l'utilisateur final soit capable de spécifier lui-

même les règles à appliquer lors de la sélection (e.g. service moins cher, celui qui correspond à sa localisation…etc).

1.2 Les principes liés au client de Widgets

Le client est une application à travers laquelle l'utilisateur consomme une ou plusieurs Widgets. Elle doit répondre aux principes suivants :

a. La composition de services native à l'environnement de travail

La capacité de composer des services (Widgets) doit être intégrée de façon native à l'environnement de travail de l'utilisateur. En d'autre terme, l'utilisateur ne doit pas avoir deux environnements distincts : un pour composer des services, et un autre pour consommer des services. Le client des Widgets doit être à la fois un environnement de travail et un environnement de composition de services.

b. Personnalisation

Il important de fournir à l'utilisateur des outils de personnalisation de l'environnement de travail (client des Widgets).

c. Découverte de services

Le client des Widgets doit s'interfacer avec le registre afin de découvrir les Widget existantes.

d. Réutilisation et composition au niveau de l'interface graphique

Comme dans SOA, la réutilisation et la composition de services est un principe essentiel dans WOA. Il important que le client des Widgets fournisse des capacités de réutilisation et de composition destinées à la fois aux développeurs et aux utilisateurs finaux. Le caractère distinctif de l'approche WOA est d'implémenter la composition au niveau de l'interface utilisateur, dans son environnement de travail. La figure 5 montre une composition de Widget au niveau de l'interface utilisateur.

Autre la réutilisation et la composition au niveau de l'interface utilisateur, il est recommandé que le client des Widgets intègre d'une part des outils de composition distribués sur différent terminaux (environnement de travail) de l'utilisateur (voir Figure 6), et d'autre part des outils de composition basés sur des données non structurées (voir Figure 7). La composition distribuée répond à la multiplicité des terminaux de l'utilisateur, et la composition basée sur les données non structurées considère la prolifération du contenu généré par l'utilisateur que ce soit dans les sites web comme Wikipédia ou dans les services conversationnels tels que la messagerie instantanée.

Figure 5 : Composition au niveau de l'interface utilisateur.

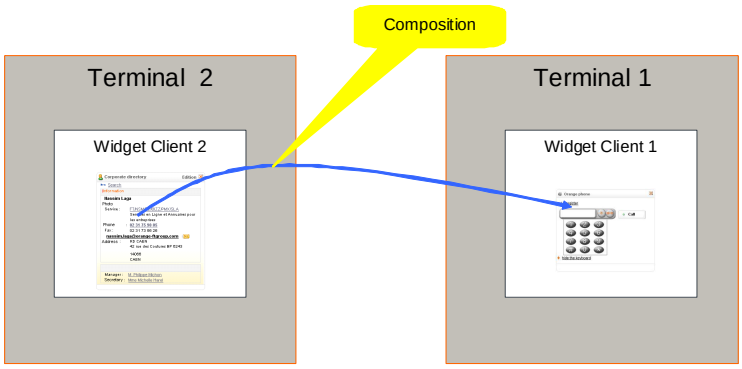


Figure 6 : Composition multi-terminal.

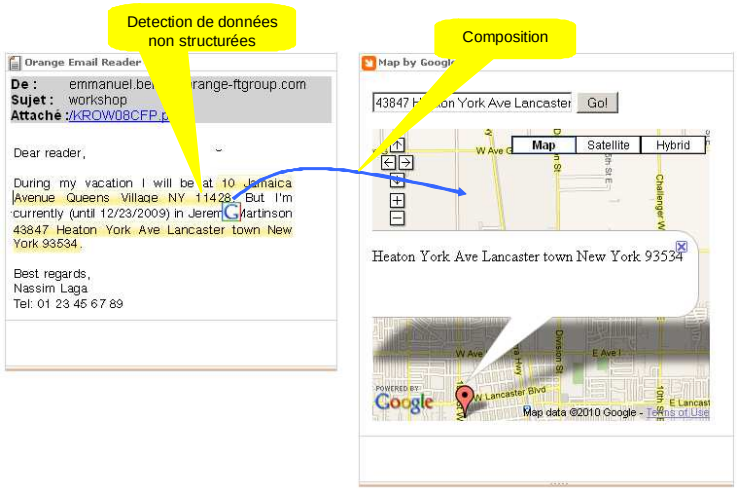


Figure 7 : Composition base sur des données non structures.

e. *Widget avec ou sans état*

Contrairement à SOA qui encourage la création de service sans état (dit aussi *stateless*), les services peuvent être avec ou sans état dans le paradigme WOA. Du moment que les services embarquent aussi

l'interface utilisateur qui interagit avec la logique métier, l'état peut être gérer au niveau de l'interface utilisateur sans pour autant affecter les performance au niveau du serveur.

1.3 Les principes liés aux développeurs et fournisseurs de Widgets

Les développeurs/fournisseurs de Widgets doivent suivre dans WOA quatre principes important:

a. *Exposition d'application sous forme de Widget*

Dans le paradigme WOA, les développeurs doivent fragmenter leurs applications en un ensemble de Widgets. Même s'il est recommandé que chaque Widget embarque une fonctionnalité, dans certain cas elle peut en embarquer plusieurs dans le but d'améliorer l'expérience utilisateur. La figure 8 montre des exemples d'applications fragmentées en un ensemble de Widget réutilisables.

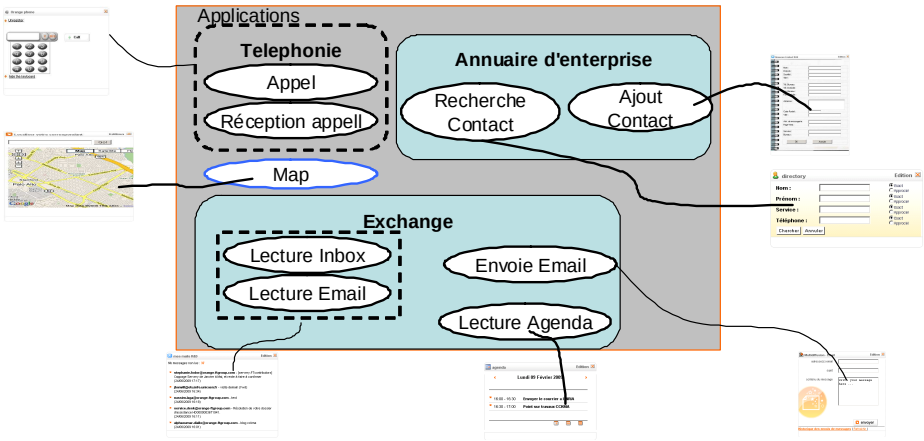


Figure 9 : Exposition d'application sous forme de Widget.

b. *Description de la Widget*

Il est important de décrire les Widgets en termes de fonctionnalités fournies d'une part, et en termes de paramètres non-fonctionnels d'une autre part.

c. *Annotation sémantique*

La composition au niveau de l'interface utilisateur nécessite que les interfaces soient sémantiquement taguées afin de permettre au client de récupérer des données générées par les Widgets pour les composer avec d'autres.

d. *Autonomie et couplage faible des Widgets*

Comme les services dans SOA, les Widgets doivent être le plus autonome possible. Elle doivent pas dépendre d'un système externe.

2 Conception du Client de Widgets

Le client de Widgets est un élément clé dans le paradigme WOA que nous proposons. Dans cette section nous allons en détailler la conception. Mais avant d'entrer dans ces détails, il est important de spécifier formellement le concept de Widget. La figure 10 montre les différents aspects d'une Widget dans notre architecture. Les éléments essentiels à retenir sont : chaque Widget fournit une description fonctionnelle et non fonctionnelle, et l'interface graphique de la Widget est taguée sémantiquement suivant le dictionnaire défini par la plateforme qui se base sur les *microformats*[6] [Khare, 2006].

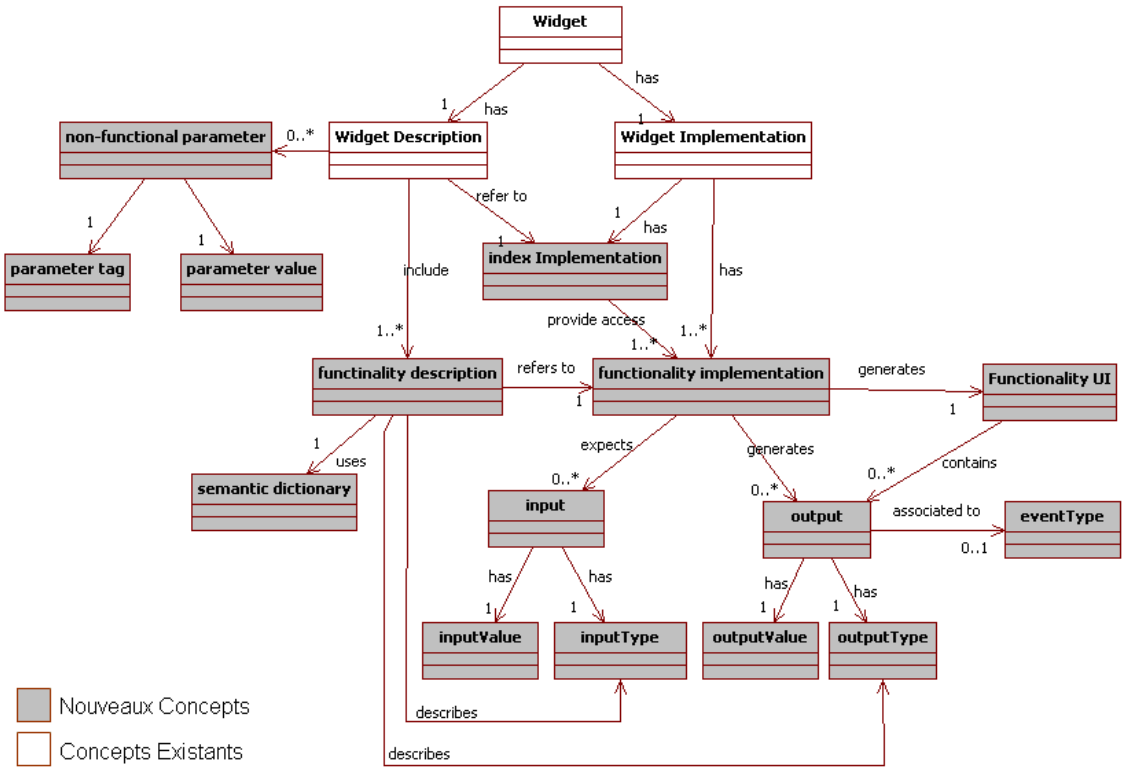


Figure 10 : Description Formelle d'une Widget.

Le client que nous proposons est un agrégateur de Widgets. En plus de l'aspect personnalisation que tout agrégateur de Widget fournit aux utilisateurs, celui-ci inclut les fonctionnalités résumées dans la Figure 11 afin de répondre à tous les principes de WOA. Les sous-sections qui suivent résument les différentes fonctionnalités et extensions.

---

[6] Microformats, http://microformats.org/, consulté le 30 sept. 2010.

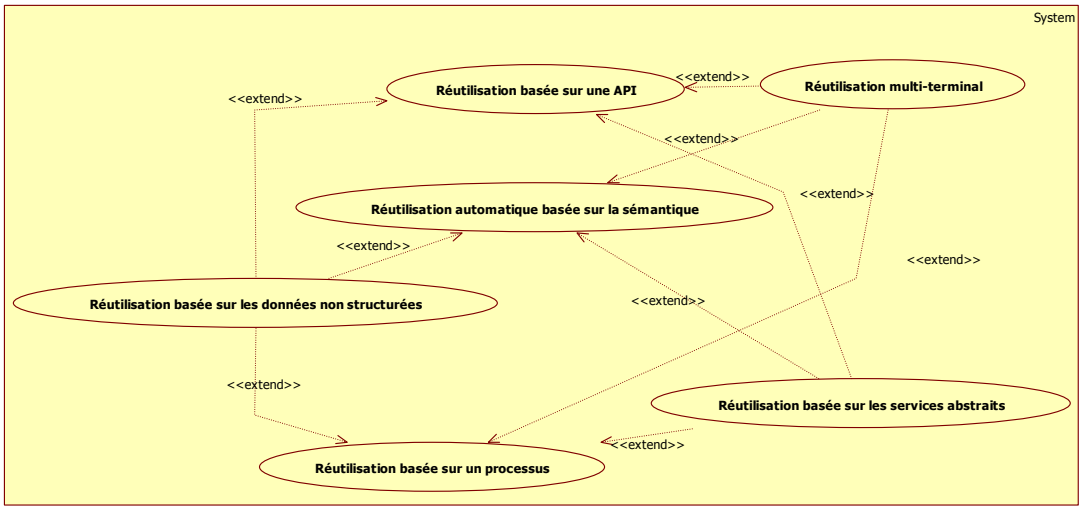


Figure 11: Les fonctionnalités clés de l'agrégateur de Widgets proposé.

2.1 Réutilisation basée sur une API

L'agrégateur de Widget offre une API qui permet au développeur d'une Widget d'utiliser les fonctionnalités d'autres Widgets lorsque celles-ci sont chargées dans la même instance de l'agrégateur. Ceci est plus orienté vers l'utilisateur final que SOA. La réutilisation de services dans SOA n'est pas limitée aux services chargés par l'utilisateur ; en réalité, les développeurs n'ont pas l'information sur les services utilisés par l'utilisateur. La figure 12 montre la différence entre notre approche basée sur WOA et les approches basées sur SOA.

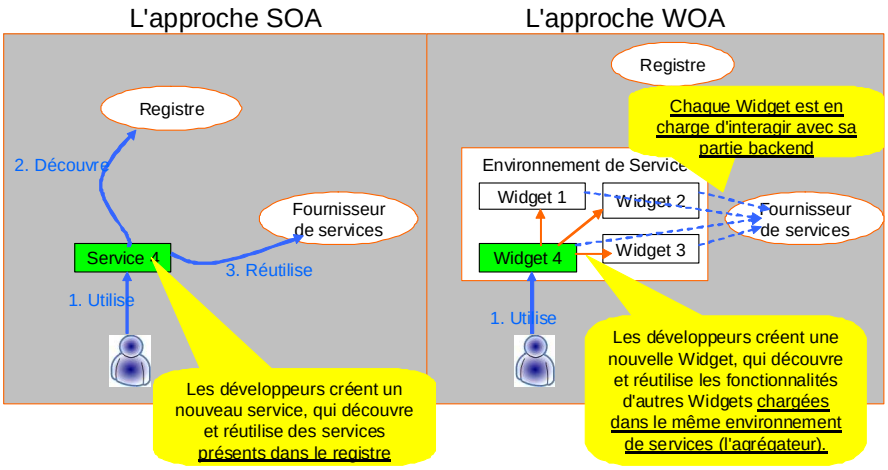


Figure 12 : Réutilisation basée sur une API

2.2 Réutilisation automatique basée sur la sémantique

La réutilisation automatique basée sur la sémantique est un mécanisme conçu pour permettre aux utilisateurs ordinaire d'assembler des services (Widgets) en fonction de leurs besoins. Basé sur les descriptions fonctionnelles des Widgets ainsi que les tags sémantiques rajoutés au niveau de l'interface utilisateur, ce mécanisme détecte automatiquement les Widgets composables et les compose au fur et à mesure que l'utilisateur les rajoute dans son environnement (agrégateur). La figure 13 montre la différence entre ce mécanisme et la réutilisation basée sur l'API que nous définissons.

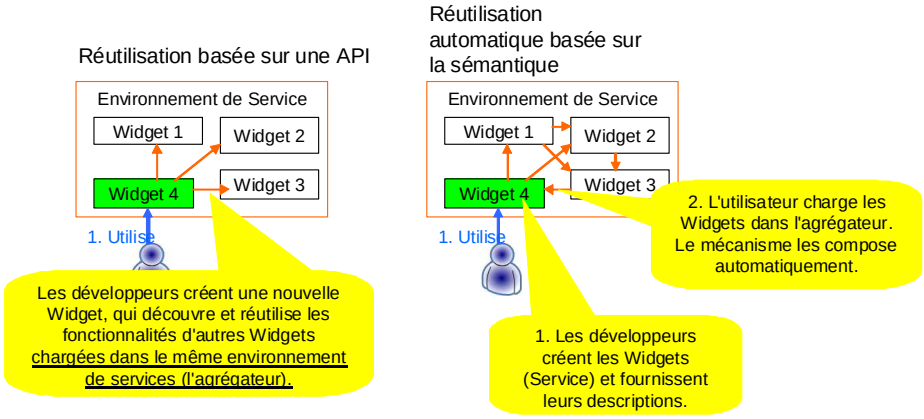


Figure 13 : Réutilisation automatique basée sur la sémantique.

2.3 Réutilisation basée sur un processus

La réutilisation automatique basée sur la sémantique est très intuitive, mais peut générer des combinaisons de services non désirées et/ou non pertinentes. La réutilisation basée sur un processus fournit à l'utilisateur la possibilité de contrôler quels sont les Widgets qui seront composées dans son environnement de services. On se base sur la définition d'un graphe spécifiant quelles sont les Widgets composées et quelles sont les données transmises d'une Widget à une autre. La figure 14 montre la différence entre la réutilisation automatique et la réutilisation basée sur les processus.

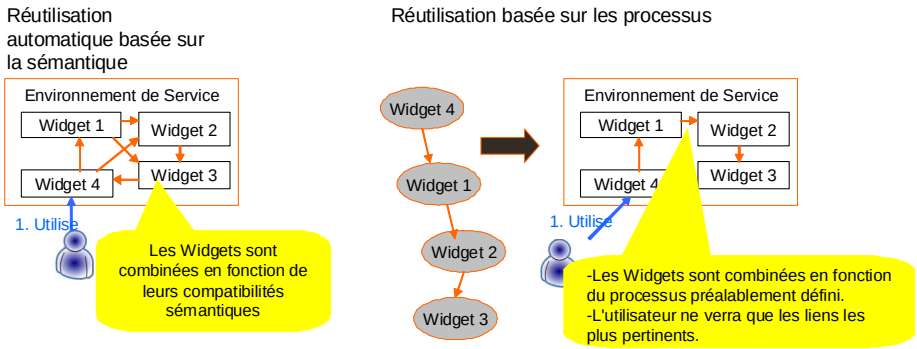


Figure 14 : Réutilisation basée sur un processus.

2.4 Réutilisation basée sur les services abstraits

Avec la prolifération des services sur le Web il est fortement probable que plusieurs services fournissent les mêmes fonctionnalités. La découverte et la sélection devient alors un challenge. Surtout lorsque les critères de sélection diffèrent d'un utilisateur à un autre et d'une fonctionnalité à une autre. Le but de la réutilisation basée sur les services abstraits est donc de fournir à l'utilisateur un mécanisme de sélection dynamique de services selon des règles spécifiées par lui-même.

Ce mécanisme de sélection orienté utilisateur est utilisé par les mécanismes décrit précédemment afin de découpler les services composés des services qu'ils utilisent d'une part, et de fournir un mécanisme d'adaptation dynamique à de nouveaux contextes selon des règles spécifiées par l'utilisateur final.

Ce mécanisme se base sur deux composants : la Widget abstraite, et l'Interpréteur. La Widget abstraite est techniquement une Widget ordinaire crée par le fournisseur de l'agrégateur et qui est associée à une fonctionnalité et un ensemble de règles applicables pour la sélection du meilleur service réalisant cette fonctionnalité. Il est important que l'interface utilisateur de la Widget abstraite permette à l'utilisateur final de choisir l'ensemble de règles à appliquer parmi celles applicables.

L'interpréteur est quand à lui responsable d'interpréter les règles afin de sélectionner le meilleur service à exécuter pour une fonctionnalité donnée. La Figure 15 résume l'architecture.

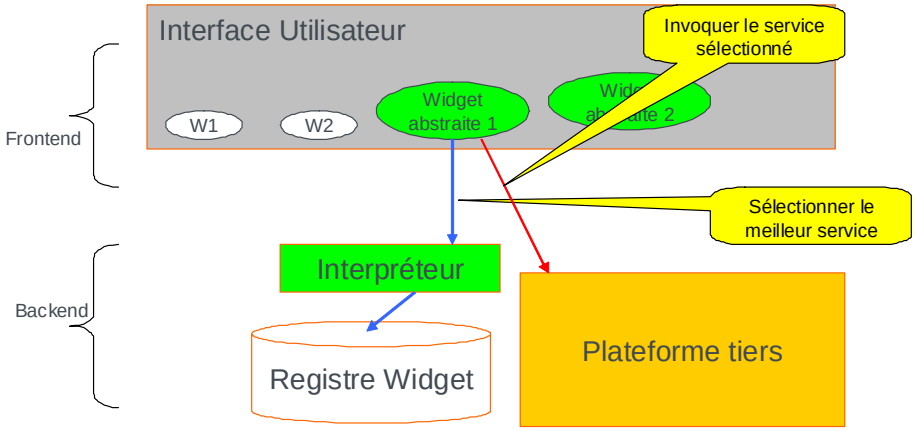


Figure 15 : Réutilisation basée sur les services abstraits

2.5 Réutilisation basée sur des données non-structurées

L'affectation d'un paramètre de sortie d'un service à un paramètre d'entrée d'un autre est sans doute la méthode la plus utilisée dans les outils de composition de service basés sur SOA. Cependant, avec la multiplication des services de communication (e.g. Messagerie, Messagerie Instantanée, Réseaux sociaux) les utilisateurs sont susceptibles d'échanger des données qui seraient pertinentes à composer avec d'autres

services. Les exemples typiques sont des adresses postales, des numéros de téléphone et des dates échangés par exemple par messagerie instantanée et qui peuvent être composés avec respectivement un service de carte géographique, un service de téléphonie, et un service d'agenda. Ce type de composition n'est malheureusement pas possible aujourd'hui en utilisant les outils de composition de services traditionnels, y compris avec ceux destinés aux utilisateurs avancés.

Le but du mécanisme que nous proposons dans cette section est de permettre ce type de composition ; à base de données non structurées. La conception de ce mécanisme est caractérisée par l'introduction d'un nouveau registre qui contient un ensemble de modules permettant l'extraction de données non structurées. Chaque module est associé à un type de données. Ainsi, au moment de l'exécution les utilisateurs peuvent associer un extracteur de donnée à une Widget. Par ce fait, à chaque fois que des données du type associé sont détectées, l'agrégateur les extrait et optionnellement les compose avec d'autres Widgets présentes dans la même instance de l'agrégateur. La figure 16 illustre ce mécanisme.

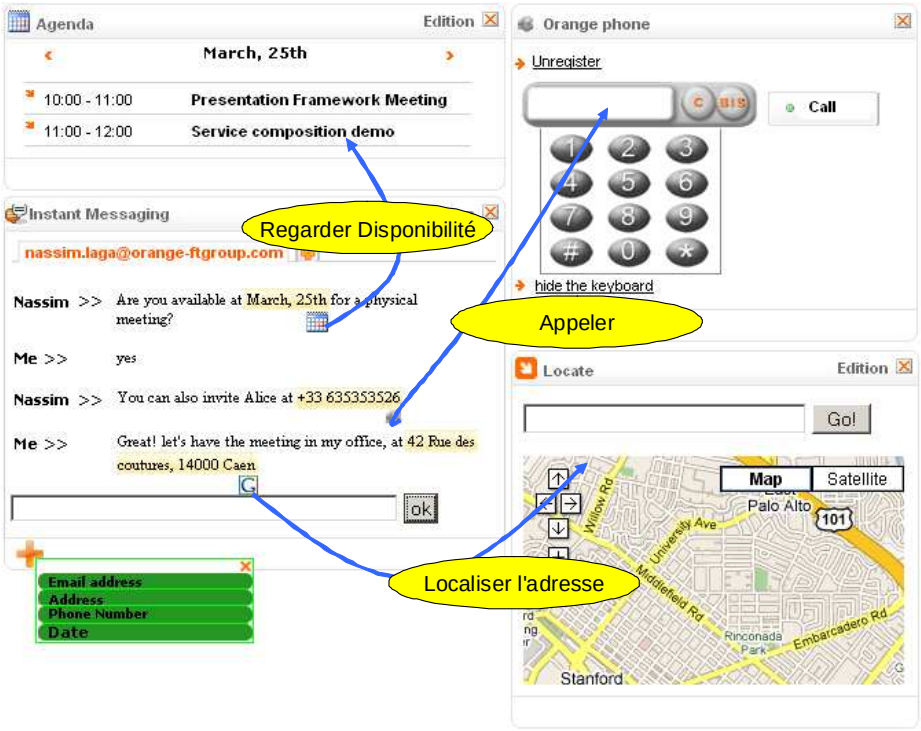


Figure 15 : Réutilisation basée sur des données non-structurées.

2.6 Réutilisation multi-terminal

Les mécanismes précédemment décrits supposent l'environnement de services de l'utilisateur limité à un seul agrégateur de Widgets exécuté sur un seul terminal. Cependant, avec la prolifération des terminaux, l'utilisateur est susceptible d'utiliser plusieurs terminaux (laptop, TV, mobile, tablette…etc.). Le

mécanisme proposé dans cette section vise à étendre les mécanismes de réutilisation de Widgets vers plusieurs terminaux d'un même utilisateur. La figure 16 illustre notre objectif.

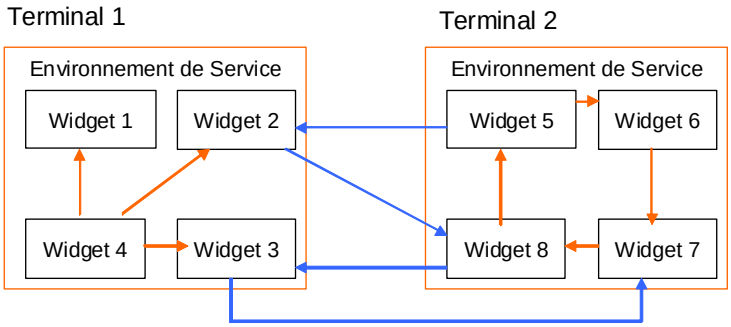


Figure 16 : Réutilisation inter-terminaux.

La conception de ce mécanisme repose sur la définition d'un protocole d'échange des informations relatives aux capacités fournies par les Widgets. Les informations sont échangées à travers une entité serveur qui fait le lien entre les terminaux d'un même utilisateur.

3 WOA dans les domaines d'application de SOA

Comme SOA, WOA est un paradigme qui peut s'appliquer à la composition de services et à la gestion des processus métiers. Dans cette section, nous allons détailler comment celle-ci sont réalisables avec le paradigme WOA.

3.1 WOA pour la composition de services

Dans l'état de l'art, nous avons classifié les outils de composition de service en trois ensembles : la composition statique, la composition semi-automatique, et la composition automatique. Nous allons donc voir comment ces approches sont réalisables avec le paradigme WOA.

La composition statique basée sur le paradigme WOA est réalisée en utilisant l'API offerte par l'agrégateur de Widgets que nous avons défini. Plus précisément, les développeurs de Widget utilisent les trois fonctions suivantes : *GetWidgetList*, *Subscribe*, et *Publish*. L'avantage de cette approche est que d'une part le développeur utilise les services utilisé par l'utilisateur final afin de créer son service composé, et d'autre part, l'utilisateur peut personnaliser un service composé en chargeant les Widgets par celles qu'ils préfèrent. La personnalisation des services composés peut se faire aussi de manière automatique en utilisant les Widget abstraite, dans lesquelles le service concret exécuté pour chaque fonctionnalité est dynamiquement sélectionné suivant des règles spécifiées par l'utilisateur final.

La composition semi-automatique basée sur SOA est actuellement délicate pour les utilisateurs ordinaires, sans connaissance en informatique. Le paradigme WOA en se basant sur la réutilisation d'interface, la sémantique, et la composition directement au niveau de l'environnement de service de l'utilisateur, permet de combler ce manque. Les utilisateurs créent un nouveau service juste en chargeant des Widgets dans leur environnement de travail. Par ce fait, il n'est pas nécessaire de connaître le concept d'organigramme, de paramètre d'entrée ou de paramètre de sortie d'un service.

Outre le ciblage des utilisateurs ordinaires, WOA permet aussi

- de composer des services distribués sur différent terminaux de l'utilisateur, de manière à ce que les différentes fonctionnalités soient exécutées dans le terminal le plus approprié ;
- de composer des services à partir de données non structurées ;
- de découpler les services composés des services de bases qu'ils utilisent (en utilisant les Widgets abstraites).

Dans l'état de l'art nous avons montré que la composition automatique dans SOA manque de précision dans le sens ou les services créés ne correspondent pas toujours et exactement aux besoins exprimés par l'utilisateur. Ceci est essentiellement du à l'ambiguïté du langage naturel. Le paradigme WOA permet à l'utilisateur d'ajuster un service créé. Cette capacité se traduit par le fait que l'utilisateur peut modifier la logique d'un service créé en chargeant et/ou en supprimant des Widget dans son environnement de travail.

3.2 WOA pour la gestion des processus métiers

La gestion des processus métiers est l'un des domaines où le paradigme SOA est largement utilisé. Cependant, comme nous l'avons détaillé dans l'état de l'art, SOA ne permet pas de répondre de façon optimale à l'hétérogénéité et la dynamicité des processus métiers d'aujourd'hui. Dans cette section nous allons définir une méthode de gestion de processus métiers basée sur WOA, afin de répondre plus efficacement à cette problématique.

Cette méthode consiste à définir un processus métier comme étant une union de deux partie : une partie commune à tous les utilisateurs, et une partie spécifique à un sous ensemble réduit d'utilisateur. La première partie est généralement stable à travers le temps, contrairement à la deuxième qui est souvent dynamique. En se basant sur ces assertions, nous proposons que la partie commune des processus métier soit modélisée par des entités spécifiques et développée comme une Widget réutilisable, et que la partie spécifique soit modélisée et implémentée par les utilisateurs eux même en utilisant les mécanismes de composition définis dans WOA. Ainsi, l'hétérogénéité est simplifiée par le fait que les entités responsables

de la modélisation des processus métiers ne s'occupent que des parties communes à une population significative d'utilisateurs, et la dynamicité des processus métiers est prise en compte car cette dynamicité concerne souvent les parties spécifiques et que celle-ci les utilisateurs finaux s'occupe eux-mêmes des modifications et des adaptations en fonction de leurs nouveaux besoins. L'adaptation des processus peut également se faire de façon automatique en utilisant les Widgets abstraites.

Implémentation et Expérimentation

Afin de montrer l'implémentation des concepts que nous avons définis, nous allons dans ce chapitre parcourir l'ensemble des mécanismes et illustrer chacun d'eux. Ces mécanismes sont actuellement utilisés dans plusieurs projets internes et externes (e.g. projet européen SERVERY, agrégateur de Widget d'Orange).

1 Réutilisation basée sur une API

La Figure 17 montre deux Widgets (Téléphonie et Annuaire d'entreprise) composés en utilisant l'API fournie par l'agrégateur que nous avons conçu et implémenté. Dans ce cas d'utilisation, à chaque appel entrant dans la Widget de téléphonie, une recherche de l'appelant est effectuée dans la Widget d'annuaire d'entreprise. Ceci est réalisé en deux étapes. Premièrement la Widget d'annuaire a préalablement déclaré qu'elle fourni la capacité d'effectuer des recherche sur un numéro de téléphone. Deuxièmement, dans le code de la Widget de téléphonie, le développeur à pris en compte les Widgets chargées dans l'espace utilisateur (en utilisant la fonction JavaScript *getWidgetList*) et publie le numéro de l'appelant à chaque appel entrant (il utilise pour cela la fonction *publish*).

Figure 17 : Composition à base de l'API de l'agrégateur.

2 Réutilisation automatique basée sur la sémantique

Le but de ce mécanisme est de composer les Widgets automatiquement au fur et à mesure que l'utilisateur les charge dans l'agrégateur. Contrairement au mécanisme précédent où les développeurs utilisent des API JavaScript (JS) pour découvrir les Widgets chargées par l'utilisateur et publier des données, le lien est ici fait automatiquement par l'agrégateur en se basant sur la compatibilité sémantique. Ce lien est symbolisé

par une icône insérée dans la Widget source de la donnée. La figure 18 montre un exemple de service composé créé par ce mécanisme.

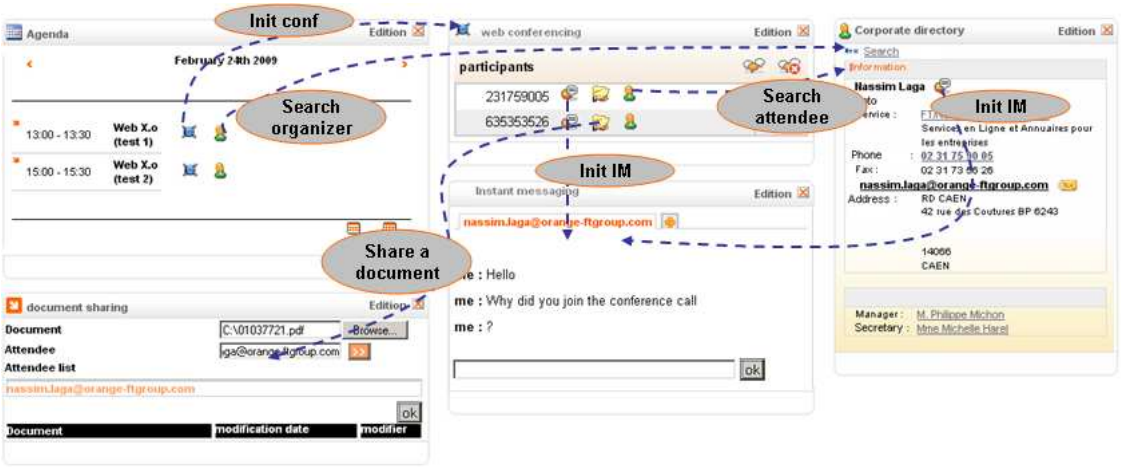


Figure 18 : Composition basée sur la compatibilité sémantique.

3 Réutilisation basée sur un processus

Le but de ce mécanisme est de fournir à l'utilisateur final un moyen de contrôler les liens entre Widgets qui sont créés par le mécanisme précédent. Pour cela, nous nous basons sur une définition d'un service composé. Ainsi, au fur et à mesure que l'utilisateur charge des Widgets dans son environnement,

- des liens sont automatiquement créés selon la compatibilité sémantique entre les Widgets,
- un processus est défini (dans sa première version, le processus contient tous les liens possibles entre les Widgets chargées dans l'environnement de l'utilisateur),

La particularité de ce mécanisme par rapport au précédent tient au fait que l'utilisateur peut supprimer des liens et en automatiser d'autre. La définition du service composé est alors modifiée en fonction.

4 Réutilisation basée sur les services abstraits

Le concept de service abstrait est caractérisé par les Widgets abstraites et l'Interpréteur. La figure 19 montre un exemple d'une Widget abstraite, dont la fonctionnalité est l'envoi de SMS. Elle permet à l'utilisateur final de spécifier des règles de sélection à appliquer sur cette fonctionnalité, de fournir les paramètres d'entrée nécessaires à l'exécution de cette fonctionnalité, et d'exécuter le service concret qui a été sélectionné par l'Interpréteur.

Dans notre exemple de la figure 19, l'utilisateur a activé la sélection selon la localisation du destinataire du SMS. Le service sélectionné est affiché en bas de la Widget.

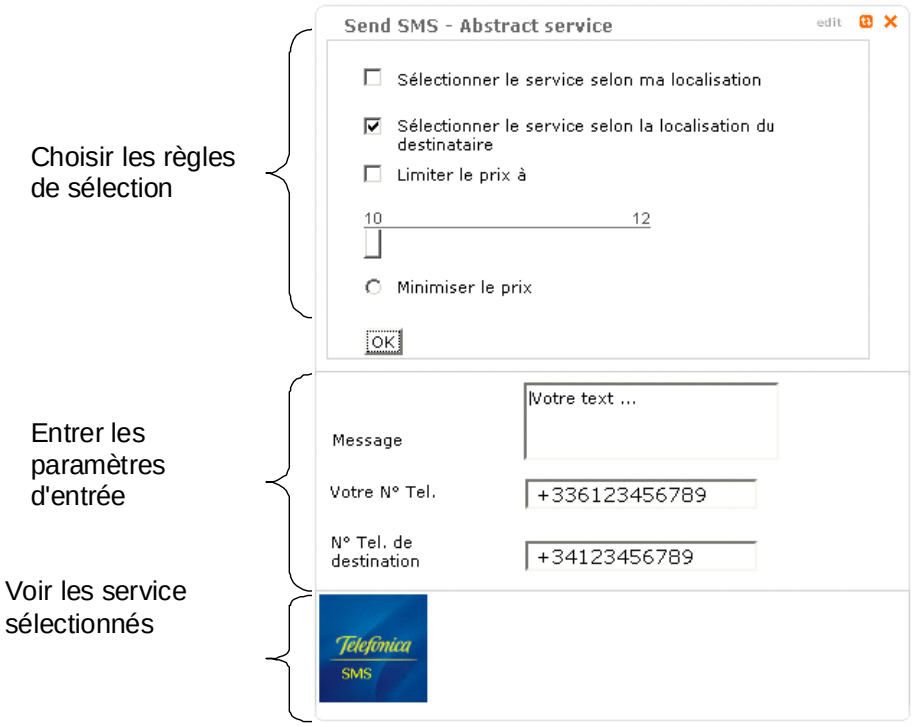


Figure 19 : La Widget abstraite.

5 Réutilisation basée sur des données non-structurées

La réutilisation de Widgets à base de données non-structurées est une architecture qui permet la définition d'enchainements de services, où les données sources sont d'abord extraites et formatées, et ensuite transmises comme paramètre d'entrée à la Widget de destination. Afin d'illustrer ce mécanisme, prenons le service composé illustré dans la figure 20. L'exécution de ce service est précédemment illustrée dans la figure 15. Ce service composé peut être créé directement par l'utilisateur final dans son environnement de services. Pour cela, il charge d'abord les Widgets nécessaires (dans notre cas, il charge la Widget de messagerie instantanée, la Widget de téléphonie, la Widget de carte géographique, et la Widget d'agenda). Ensuite, il associe des extracteurs de données non structurées à des Widgets, selon la logique du service composé qu'il veut créer (dans notre cas, on associe des extracteurs de données respectivement de type *date*, de type *numéro de tel*, et de type *adresse*, à la Widget de messagerie instantanée). A l'exécution, les extracteurs de données associés aux Widgets détectent la présence ou non des données correspondante, et optionnellement, les composent avec d'autres Widgets présentes dans le même environnement de services (figure 15).

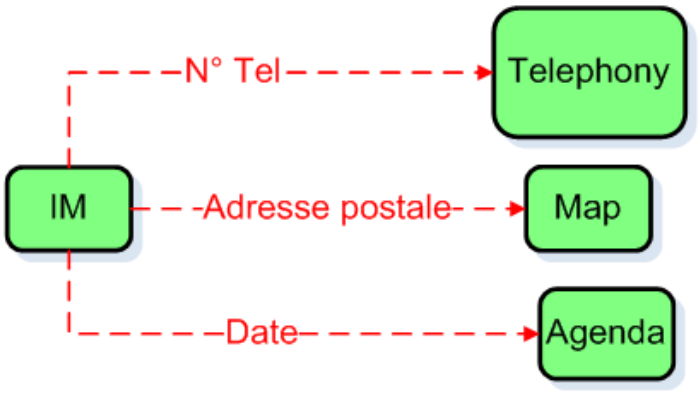


Figure 20 : service composé à base de données non structurées.

6 Réutilisation multi-terminal

La réutilisation multi-terminal permet aux utilisateurs de définir des compositions de services distribuées sur différents terminaux. Ce mécanisme repose sur la définition d'un protocole d'échange entre les différents composants que nous avons défini jusque là. Ce protocole permet à chaque composant d'avoir connaissances des Widgets chargées sur chaque terminal de l'utilisateur, et leurs capacités. Afin d'illustrer ce mécanisme, nous proposons ici deux scenarios. Le premier consiste à connecter des Widgets chargées sur deux agrégateurs tournant sur deux terminaux différents. Le deuxième, consiste à connecter une application (liste de contacts) Google Android (utilisant le protocole que nous avons défini) avec les Widgets d'un agrégateur. La Figure 21 et 22 illustrent respectivement les deux scénarios.

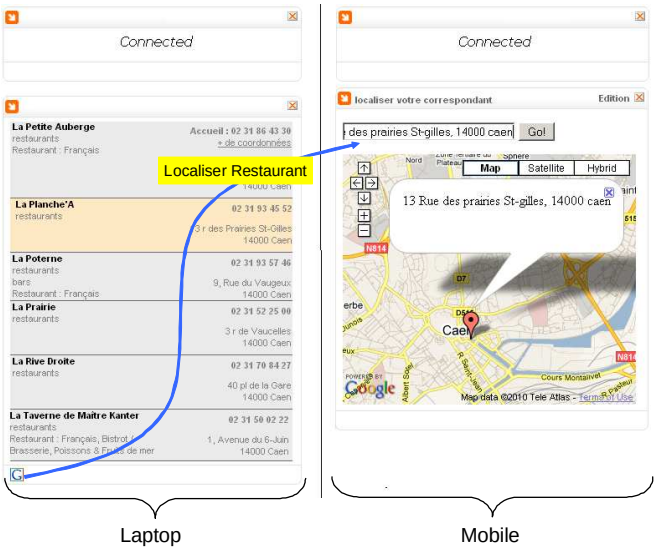


Figure 21 : Composition multi-terminal.

Figure 22 : Composition multi-terminal d'une application Android et des Widgets.

Conclusion

La plus importante contribution de cette thèse est la définition d'un nouveau paradigme de programmation orientée service basé sur le concept de Widget (WOA). Du fait de sa conception orientée utilisateur final, ce nouveau paradigme nous a permis de définir un agrégateur de Widgets qui intègre des outils de composition accessibles par l'utilisateur final.

Le paradigme WOA est caractérisé essentiellement par deux principes : le développement de services sous forme de Widgets (interface utilisateur, typage sémantique, et description de fonctionnalités), et la composition au niveau de l'interface utilisateur. Basé sur ces deux principes, nous avons défini un agrégateur de Widgets qui intègre des outils de composition au niveau de l'interface utilisateur. Nous avons défini trois approches de composition : la composition en utilisant l'API de l'agrégateur, la composition automatique et sémantique, et la composition basée sur un processus. De plus, nous avons défini trois extensions à ces mécanismes afin d'être encore plus orienté vers les utilisateurs finaux : le concept de Widget abstraite, la composition à base de données non structurées, et la composition multi-terminal. La figure 23 montre les différents mécanismes ainsi que leurs avantages.

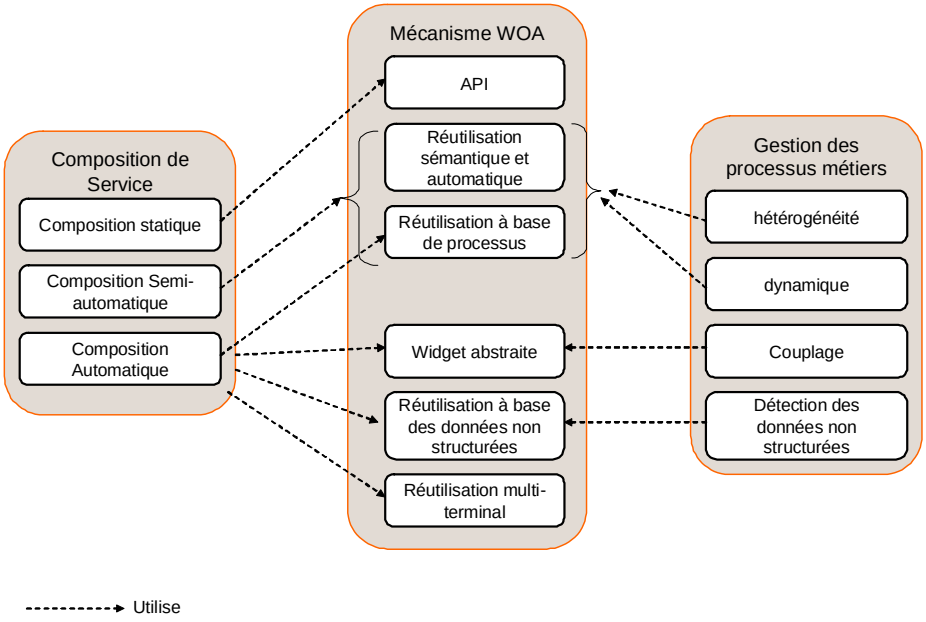


Figure 23 : Résumé des contributions.

Il important de noter que les comparaisons de WOA et SOA faites dans ce document ne préconisent pas un remplacement de SOA par WOA. Au contraire, notre but est de montrer que chacun des deux paradigmes a ses avantages et ses inconvénients, et que les deux approches doivent coexister dans une

solution globale afin de fournir des capacités de compositions de services destinées à la fois aux développeurs et aux utilisateurs ordinaires. L'approche que nous préconisons se résume en ce qui suit :

- Les développeurs créent des services web (SOA),
- Les développeurs créent les Widgets correspondantes (SOA et WOA),
- Les développeurs composent les services web, créent les Widgets correspondantes, et optionnellement, utilisent la composition statique basée sur WOA (SOA et WOA)
- Les utilisateurs composent les Widgets dans leurs environnements de services (WOA).

Les différents mécanismes introduits dans cette thèse nous ouvrent de nouvelles opportunités de recherche à approfondir. La première est l'exploitation de l'intelligence collective des utilisateurs pour enrichir des modèles sémantiques en se basant sur l'agrégateur de Widgets. Dans cette thèse nous avons utilisé un modèle sémantique peu expressif, les microformats. Cependant, la composition étant faite par l'utilisateur final, dans son environnement de services, ce manque d'expressivité du modèle est dans la plus part du temps compensé. Il est néanmoins intéressant d'approfondir cette idée pour construire des modèles sémantiques en fonction des liens entre les services créés par l'utilisateur final.

La deuxième perspective concerne la gestion des processus métiers. La découverte de processus reste un challenge, même en se limitant à la partie commune aux différents aux utilisateurs. En permettant aux utilisateurs de concevoir et d'automatiser eux même la partie qui leur est spécifique, nous pouvons facilement détecter quelles sont les pratiques les plus courantes des différentes entités d'une organisation. Par conséquent, il serait possible de concevoir des outils pour aider les entités business à décider quand des enchainements définis par des utilisateurs peuvent devenir des processus métiers à part entière.

Et enfin, le troisième sujet de recherche qui nous semble intéressant à approfondir est la composition à base de données non structurées. Dans cette thèse nous nous somme limité à l'extraction de données d'une source textuelle/HTML. Nous pensons qu'il serait encore plus intéressant d'étendre le mécanisme à des sources multimédias (voix, photo…etc.).

English Thesis

Abstract

The last decade has attracted lot of research work in Service-Oriented Computing (SOC), giving raise to standardized architectures, protocols, and technologies that enable developers to easily expose and reuse services. However, these technologies do not fully consider the users as potential actors in the creation of services based on existing ones, as advocated in Web 2.0 paradigm. In this thesis, after a deep investigation of SOC and its intrinsic SOA paradigm, we propose a new approach based on Widgets. We propose the Widget-Oriented Architecture (WOA); a new paradigm to enable a user-centric service reuse. In addition, we introduce new innovative mechanisms based on the WOA paradigm to overcome current limitations of SOA in service composition and business process management fields. This new paradigm, along with the innovative architecture and mechanisms introduced, has been validated through implementation and testing.

Introduction

Web 2.0, current “Web era”, is characterized by an increasing number of services, user participation in content creation, user centred design, information sharing, interoperability through standards, and rich user interface technologies. These characteristics have really revolutionized both software engineering methods and users interaction with software features. Software features are no longer packaged as a independent applications; instead, they are split into and published as Web services in order to promote cross-network and cross-organizations sharing, collaboration, reusability, and integration. This is known as Service-Oriented Computing (SOC) [Papazoglou, 2006] [Huhns, 2005] [Casati, 2007]. For instance, Major Internet companies such as Yahoo, Google, and Amazon provide to their customers and to third party developers reusable services such as Online Storage, Email, and Maps; an approach which is also adopted by telecom operators, renaming it "telco 2.0", where functionalities such SMS, MMS, Localization, and Telephony are exposed to third party developers on the one hand, and third party services such as Maps and social networks are used within telecom applications on the other hand.

User interactions with software features have also changed during the transition from Web 1.0 to Web 2.0. This is characterized by replacing traditional desktop applications by remote and on demand applications from one hand, and providing an active role to the users in the evolution of services from another hand. First, Web based applications are more and more complete and mature, and they progressively replace traditional desktop applications in the user daily life. Even hardware capabilities such as storage and computing can be sold and bought on demand through the Web. This is known as XaaS, which refers to “Everything as a service” (Software, Infrastructure, Platform, Communication…etc.). Second, users are no longer considered as “pure” consumers of services, but instead they play a prevailing role in testing and evolving these services. Typical examples of this phenomenon are Wikipedia[7] and YouTube[8] web sites where their success is completely dependent on the quality and the quantity of content generated by users. User participation in content creation is definitely a success approach as for example YouTube web site has reached in January 2009 100.9 million of U.S. viewers according to Comscore[9].

These two characteristics of the current Web platform, namely software fragmentation into Web services and user participation in content creation, have encouraged the idea of enabling the user to create

---

[7] Wikipedia, http://en.wikipedia.org/wiki/Main_Page, accessed Dec 22, 2009
[8] YouTube, http://www.youtube.com/, accessed Dec 22, 2009
[9] Comscore, statistics on online videos, http://www.comscore.com/Press_Events/Press_Releases/2009/3/YouTube_Surpasses_100_Million_US_Viewers/%28language%29/eng-US, accessed Dec 22, 2009

and publish new services by composing existing ones. This is known as user service creation. The goal is to harness the user intelligence in service creation in the same way we do it today (Web 2.0 era) in content creation. Consequently, several approaches have been proposed. Some of them are automatic such natural language based service composition, and others require additional investment from the user by manually and graphically chaining services.

To access and consume services, users rely on what we call in this thesis "user service environment". This term covers any software application that enables the user to access, manage and consume services. This includes traditional desktop environments (e.g. Microsoft Windows, MAC OS, and Linux), emerging Web-based operating systems (e.g. Google Chrome OS, Wiki-OS[10], Glide OS[11], and eyeOS[12]), and Widget aggregators (e.g. iGoogle[13] and Netvibes[14]). Historically, there are two important evolution phases of user service environments. The first one is characterized by evolving from CLI (command line interfaces) into WIMP (Windows, Icons, Menus, and Pointer) interfaces, and the second one is characterized by considering the XaaS paradigm where software applications are hosted and running remotely in the Web. Even the user service environment is sometimes hosted remotely in the Web (e.g. Wiki-OS, Glide OS, and Netvibes); this implies a significant simplification in term of storage and computing capabilities of users' devices, as well as in term of software maintenance. In addition of adopting the XaaS paradigm, Widget aggregators attempt to replicate the technical and the conceptual evolution from Web 1.0 to Web 2.0. Indeed, from the technical perspective, Widget aggregators follow the Web 2.0 fragmentation into Web services. They rely on the concept of a Widget to access and use functionalities available remotely on the Web. From the conceptual perspective, Widget aggregators follow the user-centric design. They promote personalization by enabling users to create their own Mashup by loading different Widgets of different providers on the same environment. However, while Web services, SOA (Service-Oriented Architecture), and SOC have attracted much academic attention during these two decades, Widget paradigm is not really investigated. This thesis aims to fill this gap. It aims not only to study the impact of the Widget paradigm on existing technologies of SOC, but also to attempt to succeed by using Widgets where SOA failed.

[10] Wiki-OS, https://www.wiki-os.org/, accessed Dec 22, 2009
[11] Glide-OS, https://desktop.glidesociety.com, accessed Dec 22, 2009
[12] eyeOS, http://www.eyeos.com/, accessed Dec 22, 2009
[13] iGoogle, http://www.google.com/ig?hl=en, accessed Dec 22, 2009
[14] Netvibes, http://www.netvibes.com, accessed Dec 22, 2009

1 Problem Statement

Service creation methods have significantly evolved since the beginning of computing technology. The starting point was assembly languages; low level programming languages that enable developers to define a sequence of instructions to be performed by a microprocessor. Then, in order to accelerate service creation and reduce the time to market, the philosophy of reusability progressively took momentum in the evolution of service development methods. Indeed, developers firstly have in their disposal macros and functions which are sets of instructions that are reusable within the same program. Thereafter, the concept of object has emerged in Object-Oriented programming OOP [Cox, 1986], which is reusable over different applications. We also retrieve the concept of object in CORBA (common object request broker architecture) [Vinoski, 1997] and OLE[15] (Object Linking and Embedding). CORBA is cross-network and programming language independent service architecture and OLE is a Microsoft technology that enables applications to exchange data with each other. Finally, in SOA, reusable service and/or reusable resource have respectively emerged in Web services architecture (WSA) [Newcomer, 2002] and REST (Representational State Transfer) [Fielding, 2000]. SOA has attracted much attention during this last decade, giving rise to several research and application frameworks that cover heterogeneous fields such as Service Composition, Business Process Management, and Pervasive Applications. However, the listed service computing technologies, as well as the emerging ones in the Web, do not fully satisfy Web 2.0 best practices, where:

- the user is placed at the centre of service development (user centricity),
- personalization becomes more important than ever due to the heterogeneity and dynamicity of user needs,
- user interface (UI) and user experience are important criteria in service development.

The goal of this thesis is to define a new paradigm where the listed best practices are fulfilled by:

- enabling users (without computing skills) to modify existing services, or creating new ones through composition according to their needs, and thud promoting personalization,
- considering the UI as an important criterion while enabling the personalization and the creation of services,
- considering the proliferation of user devices, and the need for composing services loaded on different devices,

[15] Microsoft OLE, http://msdn.microsoft.com/en-us/library/aa271010%28v=VS.60%29.aspx, accessed on July 31, 2010

- considering session based services (e.g. telephony and IM), where the capabilities may change from a state to another (e.g. the user is connected, a communication is established…etc),
- considering the proliferation of user generated data, which could be useful for composition with other services (e.g. extracting postal addresses from an IM discussion and composing it with a Google Map service),
- considering users own criteria in service discovery.

2 Contributions

This thesis aims to first introduce an alternative to SOA that satisfies the listed requirements; and second to formalize and validate it by considering different SOA application fields such as Service Composition and Business Process Management. Therefore, we first introduce a Widget-Oriented Architecture (WOA); a new computing paradigm that aims to be more user centric (Section 2.1). Then, using this new paradigm, we explore two SOA application fields, namely Service Composition (Section 2.2) and Business Process Management (Section 2.3).

2.1 Widget-Oriented Architecture (WOA)

The WOA relies on Widgets in the development of software features. A Widget is a small client-side web application for offering atomic functionalities of a software feature. A Widget includes a UI for each operation of a software feature. Thus, when a service provider creates a service, he also associates a UI that facilitate the user-service interaction. This enables to create rich UI and provide something more meaningful for ordinary users, instead of reading XML files, which are addressed more for developers.

The WOA is basically characterized by a Widget developer, a Widget provider, a common Widget registry, the user, and the Widget client which embeds the necessary mechanisms that enable the user to combine the Widgets.

2.2 Service Composition using WOA

Service reusability and composition are the driving concepts of SOA. In this thesis we have classified them into three categories: static service composition, automatic service composition, and semi-automatic service composition. In summary,

- static service composition aims to provide technologies to developers to perform reusability and composition,
- automatic service composition aims to enable users to generate services by expressing their needs using their natural language,

- and semi-automatic service composition aims to enable users to create a composite service by graphically chaining ready-made services.

However, while SOA has succeeded in static service composition, the automatic and semi-automatic ones still suffer from several limitations detailed in *Chapter I.1 State of the Art*. In this thesis, we demonstrate the potential of the WOA paradigm regarding the static, automatic, and semi-automatic service composition. First, we enable a user-centric static service composition using Widgets. Second, we significantly enhance the intuitiveness of semi-automatic service composition by introducing new approaches, centred on the users. Third, we enhance the performances of automatic service composition by enabling users to intuitively refine a created service; which does not always mach exactly the user needs as automatic composition still suffers from its inaccuracy. In other words, we use the user service composition capability as a failure recovery system in automatic composition. Finally, we enrich these three composition tools with three concepts:

- abstract service based composition in order to first decouple composite services from the basic services they invoke; and second, take into account user context and preferences in the execution of the composite services;
- unstructured data based composition in order to consider within the composite service definition data that are not declared nor formatted by developers at the service publication time;
- multi-device service composition in order to take into account the proliferation of user devices, and provide the capability of composing services loaded on different devices.

2.3 Business Process Management using WOA

SOA has significantly improved business process management and integration. However, due to developer centricity of the SOA paradigm, business process management and integration still do not provide the flexibility needed by users and business process managers. This is due to the complexity of business process definition tools from one hand, and the difficulty for service developers to detect all data that might be useful in performing a business activity. These two points are detailed in the state of the art *Chapter I.1 State of the Art*.

This thesis demonstrates the potential of WOA to tackle the two limitations in managing and integrating business processes. First, by enabling the ordinary users to compose services using the Widget paradigm, we also enable users to personalize their business processes. This tackles the heterogeneity as well as the dynamicity of business processes. Second, we introduce the dynamic adaption of composite services, which enables developers to implement business processes that automatically adapt their behaviour according to a new context. Third, we propose and validate a new mechanism named

"unstructured data based service composition". This mechanism enables users and developers to chain two services based on data that are not expected as legacy outputs of the first service. From the business analyst perspective, this enables capturing unstructured data which circulates between employees (e.g. postal addresses within an emails), and from the technical perspective, this alleviates service developers from annotating and formatting data which are hardly expectable as legacy outputs of their services.

3 Context of the Thesis

This thesis is carried out mainly at Orange Labs (Business Unified Communication (BUC) laboratory), and Telecom SudParis (Réseaux et Services Multimédia Mobiles (RS2M) department). It is supported mainly by two projects: namely CCKMA and SERVERY. CCKMA, for *Communication, Collaboration, Knowledge, and Mobile Access*, is an Orange Lab internal project which aims to study and define innovative solutions for communication and collaboration within business organizations context. The second project, SERVERY, for *Advanced SERVice Architecture and Service DelivERY Environment*, is a Celtic European project[16] that aims to build a marketplace of converged services (Telecom and Web services), where service creation, service management, and their execution on heterogeneous platforms is supported.

4 Manuscript Organization

This manuscript is divided into three parts. The first part reviews the SOA paradigm (including service composition and business process management), the evolution of service environments, and the Widget and Widget aggregator (a user environment type) paradigms. The second part of this thesis details our main contributions. It includes a Chapter which introduces the principles of the WOA paradigm, a Chapter which details the design of a framework (Widget aggregator) compliant to these principles, and a Chapter for defining how WOA is applied respectively in service composition field and business process management field. The third part of the thesis details the implementation of our contributions. It includes a Chapter for detailing the implementation of WOA, a Chapter for illustrating how it is applied respectively in service composition field and business process management, and a Chapter for summarizing our experimentations. Finally, we conclude with a summary of our contributions, the advantages of WOA compared to SOA, and some future research directions regarding this field.

[16] Servery project, http://projects.celtic-initiative.org/servery/, accessed Dec 22, 2009

Part I State of the art

This part investigates the service-oriented computing and the Widgets paradigms. We highlight the advantages and limitations of each one in order to propose, in the second part, a new architecture paradigm that includes the best of each of them.

Chapter I.1 State of the Art

The general context of this thesis is halfway between service environments and service-oriented computing (SOC). Therefore, in this state of the art Chapter, we will investigate the concepts related to these two fields. We clarify in the first section the definition of the term Service. Then, we investigate the SOC paradigm in section 2. This includes the study of Service Oriented Architecture (SOA) and related technologies, and the investigation of service composition and business process management fields. In section 3, we study current service environment approaches. We detail the Widget paradigm in section 4. We summarize in section 5 current Web semantic technologies, as this field is omnipresent in current Web 2.0 and software engineering research. Finally, we conclude with the limitations of all these fields.

1 Services

In order to clarify the meaning of the term *Service*, we study in this section the different definitions that different entities assign to this term. We start by providing the user view, and then, we study the technical usage of the term. Literally, the term Service refers to several meanings. Followings are an enumeration of some selected definitions from Collins dictionary[17].

- *An act of help or assistance,*
- *An organization or system that provides something needed by the public, a consumer information service,*
- *the installation or maintenance of goods provided by a dealer after a sale,*
- *the serving of guests or customers,*

The first definition is generic; it includes any act of help or assisting someone or something. Instead, the others are more focused and define the service as a relationship between a provider and a consumer (person). Indeed, the second definition defines a service as a relationship between an organization, or a system, and the public. The third one defines the service as a relationship between a dealer and a consumer (person). Finally, the fourth one defines a service as the act of serving a guest or a customer.

In IT (Information Technology) and Telecom communities, we also find several meaning of the term Service. Some of them differ, and some others converge. Table 1 summarizes the different definitions of the term according to different entities (Standardization entities and European projects). We distinguish

[17] http://www.collinslanguage.com/results.aspx, accessed Dec 22, 2009

two main orientations. The first one includes SPICE [Cordier, 2006], OPUCE [Yelmo, 2008], and OMA [OMA, 2007] definitions. They define a service through its usage properties. They highlight the relationship between a provider and a consumer. The second orientation includes the SeCSE [Sawyer, 2005] and W3C [W3C, 2004b] definitions. They define a service through its technical properties. They highlight the fact that a service is a software entity, and it is described at least through one service description. When aggregating both approaches, we can deduce that a service is a software entity that performs one or more operations. It is developed by a service developer and has at least one service description. It is made available by a provider and consumed by a consumer, who is optionally charged for. This definition highlights three roles: the developer, the provider, and the consumer. While the developer and the provider are obviously human entities, the consumer is ambiguous. Indeed, the consumer of a service might be a user that invokes the service in order to consume it, or a developer who may reuse the service to create a more innovative one. This distinction is introduced in OMA. Indeed, they distinguish between software entities which are consumed by another software entity, and those that are consumed directly by users. Thus, the former is named *Enabler* while the latter is named *Service*.

Table 1. Technical definitions of the term Service.

Entity	Definition
SPICE [Cordier, 2006]	*A service is an added value that is provided by a service provider to an end user.*
OPUCE [Yelmo, 2008]	*Provider-client interaction that provides value. Service properties are described with a service specification.*
SeCSE [Sawyer, 2005]	*Software entity that performs one or more Operations. It is developed by a Service Developer and has at least one Service Description.*
OMA [OMA, 2007]	**Enabler:** *A technology intended for use in the development, deployment or operation of a Service; defined in a specification, or group of specifications, published as a package by OMA.* **Service:** *A selection from the portfolio of offerings made available by a service provider, which the user may subscribe to and be optionally charged for. A service may utilize one or more service enablers.*
W3C [W3C, 2004b]	o *An application that provides computational or informational resources on request. A service may be provided by several physical servers operating as a unit.* o *A service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities.*

2 Service-Oriented Computing (SOC)

Service-Oriented Computing (SOC) [Papazoglou, 2006] [Huhns, 2005] [Casati, 2007], or Service-oriented programming (SOP) [Sillitti, 2002] [Bieber, 2001] is "*a new computing paradigm that utilizes services as the basic constructs to support the development of rapid, low-cost and easy composition of distributed applications even in heterogeneous environments*" [Papazoglou, 2006]. This definition highlights well that SOC is more a vision than a technology [Margaria, 2007]. It is characterized by two roles: service provider, and service consumer. The former creates services and makes them available for third parties, and the latter invokes the service when needed. However, before using a service, consumers must discover it. This might be performed in an ad-hoc way, or using the facilities that are provided by Service-Oriented Architecture (SOA) paradigm. SOA introduces a mediator role who is in charge of collecting available services from providers and making them available for consumers.

As Bichler and Lin stated in [Bichler, 2006], SOC paradigm has evolved from earlier component-based software frameworks such as Enterprise JavaBeans (EJB) [Thomas, 1998] and Common Object Request Broker Architecture (CORBA) [Vinoski, 1997]. Currently, SOC is mainly performed using Web Services Architecture (WSA). Firstly, because WSA is strongly supported by major software companies such as *IBM*, *Microsoft*, *Hewlett-Packard*, *Oracle*, and *SAP*. Secondly, because Web Services rely on openly available Internet protocols such as HTTP and XML; exploiting by this way the Web as a transport media [Huhns, 2005]. Finally, because the related technologies such as SOAP, WSDL, UDDI, and BPEL, have been standardized, as they are largely supported by major software companies.

This evolution to SOC is the result of several years of research and best practices in heterogeneous fields [Papazoglou, 2006], namely: software engineering, telecommunication, and business process management. Reusability, loose coupling, abstraction, and virtualization have been the driving needs in this evolution.

In software engineering the reusability and abstraction started with the assembling languages, when microprocessor specific directives hide the binary code, and macros and functions enable the reuse of a sequence of instructions. Then, advanced languages (sequential or object oriented [Dahl, 1968], [Goldberg, 1976], [Cox, 1986]) have emerged (e.g. C, Simula, smallTalk, and C++). Though they provide advanced directives, that wrap a set of assembling language instructions, they still depend tightly on the target hardware and operating system. The success of Java language is essentially due to its abstraction of the target hardware and operating system. It enables the portability of applications through different machines. In the 1990s, cross network reusability tools have emerged (e.g. CORBA and DCOM [Grimes, 1997]). The first version of CORBA specification is released in 1991, the current version is available at

[OMG, 2008a]. Finally, in the beginning of 2000s, WSA and REST architecture have emerged, and have been largely adopted. The driving needs have always been accelerating software creation process. This is achieved by promoting reusability and abstraction. Indeed, functions are reused in the sequential programming. Objects are reused in OOP and CORBA. Finally services (resp. resources) are reused in WSA (resp. REST based architecture).

In telecommunication field, the idea of abstracting the network capabilities started in the 1980s [Magedanz, 2007] in order to decouple the creation of new telecommunication services from the network components [Bertin, 2009]. The Intelligent Network (IN) has then emerged as the enabling architecture and technology. The IN architecture is characterized by extracting the service intelligence from the legacy network switches. A protocol was then defined and standardized on top of Signalling System 7 network for this purpose: IN Application Protocol (INAP). The standard defines the way the IN entities interact. For instance, it enables the service control point (SCP), where the service logic is implemented, and the service switching point (SSP) to interact with each other. The service logic is implemented as a chain of capabilities, so-called Service Independent Building Blocks (SIBs). A limited set of reference SIBs have been defined by the ITU-T, grouped under the name of capability set (CS-1, CS-2, and CS-3). However, as stated in [Magedanz, 2007], this limited set of SIBs limits the functional capabilities of IN. In addition, despite the wide adoption of the INs all around the world, the envisioned open market of SIBs was not enabled.

In the beginning of 1990s, the convergence between telecommunication and IT began to happen, and their evolution took more or less the same path. Indeed, telecommunication started to use IT technologies for rapid and low cost service creation. They have used for instance object oriented programming techniques and RPC-like tools (e.g. Parlay API and Java APIs for Intelligent (Integrated) Network (JAIN)). They have also used scripting languages (e.g. Call Processing Language (CPL) RFC 2824 and RFC 3880, SIP CGI [IETF, 2001], VoiceXML [W3C, 2004f], CCXML [W3C, 2010]) for rapid creation of advanced telephony services from scratch (e.g. Call forward on busy/no answer and Call Screening). The scripts can be created manually, or using graphical tools (e.g. MetaEdit+[18]), known as high level service creation tools [Glitho, 2003]. The scripts are stored in the signalling servers (e.g. H323 gatekeeper, SIP proxy, SIP redirect, and SIP registrar) and associated to source/destination addresses. When a call establishment request arrives, the signalling server detects if there is an associated CPL script, and runs it. When Web services technology has emerged in the IT community, the telecommunication community also adopted it by developing the Parlay X API based on Web services. Finally, current success

[18] MetaEdit+, http://www.metacase.com/, accessed on October 19th, 2010

of REST APIs (thanks to its simplicity), motivated telecom operators to publish their capabilities through REST APIs (e.g. Orange Partner[19] and Deutsche Telekom Developer Garden[20]).

In business process management field however, the driving needs are essentially the loose coupling between different entities, and the flexibility of business processes. The advantages brought by SOC to this field are essentially the separation between service description, service invocation method, and service implementation. Thus, business process integrators can easily discover and invoke a third party service without having any knowledge about its real implementation. In addition, the reusability of existing services makes adaptation to new requirements easier, as the implementation of a new business process is much faster; though it is still in charge of developers. Indeed, though graphical tools such as Eclipse BPEL designer[21] for modelling business processes have emerged, they still remain understandable only by developers as we detail in the following subsections.

In the following subsections, we first detail the Service-oriented Architecture (SOA) from the conceptual and the technical perspectives; SOA is considered as the enabling architecture for SOC. Then, we illustrate how service composition and business processes management research benefit from SOC. Finally, we conclude with current unresolved issues.

2.1 Service-Oriented Architecture (SOA)

SOA is an architecture that enables SOC. SOC paradigm is characterized by two entities: service providers who expose their services, and service integrators who reuse these services in the development of their own service or application. For a wide adoption of SOC, discovery and publication facilities must be provided respectively to service integrators and service providers. SOA is an architecture paradigm that provides such facilities. It provides a centralized approach for enabling service providers to publish their services, and service consumers to discover them. This assertion is supported by W3C definition of SOA [W3C, 2004a]: "*A set of components which can be invoked, and whose interface descriptions can be published and discovered*." As we illustrate in Figure 1, SOA is based on a common registry where service descriptions are stored. Though this registry might be physically distributed over several platforms (such as [Du, 2006], [Verma, 2005], and [Podesta, 2008]), it is still remain a central entity from service providers and service consumers points of view.

---

[19] Orange Partner, http://www.orangepartner.com, accessed on October 19th, 2010
[20] Deutsche Telekom Developer Garden, http://www.laboratories.telekom.com/ipws/English/News/Presse/2010/Pages/DeveloperGarden.aspx, accessed on October 19th, 2010
[21] Eclipse BPEL designer, http://www.eclipse.org/bpel/, accessed on July 31, 2010

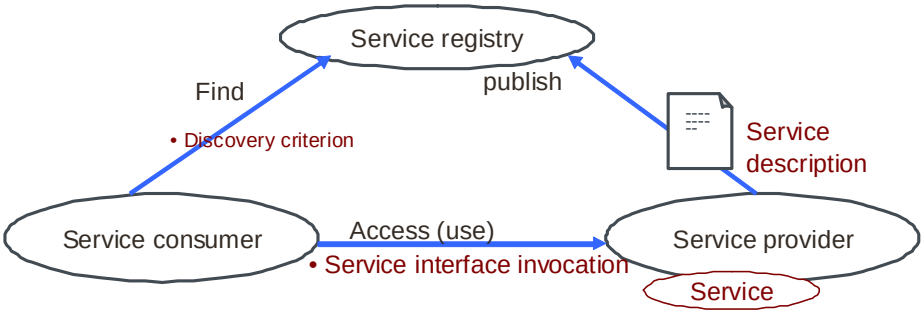


Figure 1: Basic service oriented architecture.

The SOA paradigm is based a set of principles [Erl, 2007]; some of them concern the service developers and providers, while others are related to the service registry. Thus, service developer and provider should:

- describe their services (service contract) in term of the functionalities they provide and their non-functional properties such as (price, Qos, SLA, version of the service,…etc),
- hide to the external world the complexity of the underlying implementation of the service logic (service abstraction),
- reduce as much as possible the dependencies with other services (service autonomy),
- create stateless services.

The service registry should provide interfaces for publication and discovery of services.

In addition to these principles, service reusability and composition is a main principle that should be ensured by a service oriented system.

Web Services Architecture (WSA) is currently considered as the intrinsic technology of SOA. However, there are actually several other technologies that enable it, namely OLE, EJB, CORBA, and REST. In addition to Web services, we detail in the following subsections these technologies. We demonstrate and explain that current technologies are centred on the developer needs, and do not consider users as potential creators of advanced services from existing ones..

a. OLE Automation

OLE (Object Linking and Embedding) automation is a Microsoft technology embedded natively in Microsoft Windows desktop environment. It enables one application to discover and use capabilities of another one. The concept follows a SOA as depicted in Figure 2.

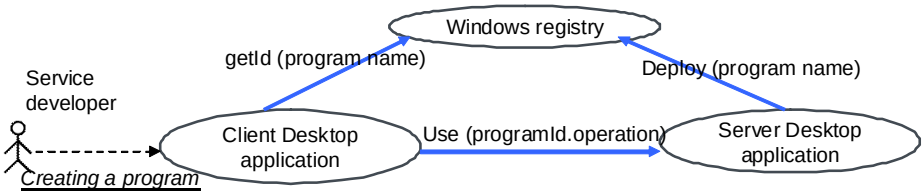


Figure 2: OLE automation.

When a developer creates an application, he specifies whether his application can behave as a server (i.e. it can be accessed and controlled from other applications). Consequently, when users install this application in their desktop environment, it registers itself in the windows registry. Thus, other developers, who are aware about this application, its operations, and their availability at the runtime, can create an application that requests the identifier from the registry and use the application specific operations as depicted in Figure 2.

b. Enterprise JavaBeans (EJB)

EJB [Sun, 2001] is a *Sun Microsystems Inc*. architecture and technology for the development of component-based applications. In this section we summarize the main concepts of the architecture. As we illustrate in Figure 3, the architecture includes mainly three roles: a Bean provider, an EJB server and container, and an application assembler. The Bean provider is the entity which exposes its software features to third parties. It first implements the business logic of its software. Second, it defines two interfaces: Home Interface and Remote Interface. The former enables the Application Assembler entity to create and remove instances of the Bean, and the latter enables them to call the Bean specific methods. Third, the Bean provider must implement the session Bean interface; an implementation which includes public methods which will be used as entry points to access to the business logic of the software feature. Fourth, the provider associates a name to the Bean and deploys it to an EJB server, using a deployment tool. The deployment tool automatically generates implementations for the two (Home and Remote) interface definitions; implementations which will be used by the Application Assembler to create instances of the published Bean and invoke its methods.

The Application Assembler is the consumer of a Bean. It first discovers an existing Bean within an EJB server using Java Naming and Directory Interface (JNDI) API [Sun, 1999]. Then, it uses the Home interface to create instances, and the Remote interface to invoke methods (Home and Remote interface are previously generated when publishing the Bean).

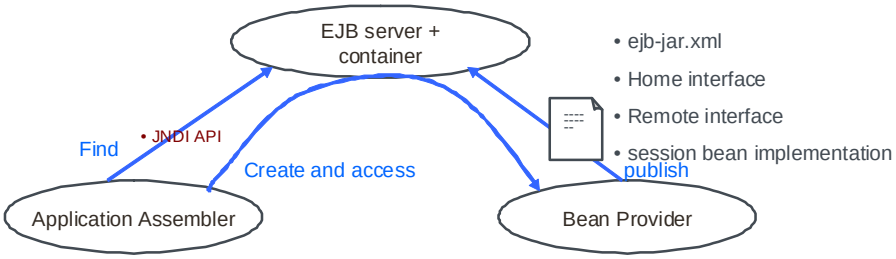


Figure 3: EJB-SOA analogy.

EJB architecture does not really conform to service-oriented model. First, EJB is limited to Java language and J2EE environment. Second, all invocations to a Bean go through the EJB server. This implies that there is always a central entity which bridges between different stakeholders (provider and consumer). While this approach may succeed within a limited environment such as a company, it is still not scalable enough to be widely adopted within the Web platform.

c. Common Object Request Broker Architecture (CORBA)

CORBA [OMG, 2008a], Common Object Request Broker Architecture, is an Object Management Group (OMG) standardized architecture that aims to facilitate the development of distributed applications. It enables for instance the development of loose coupled objects which communicate with each other. It enables a developer to invoke in his applications ready-to-use objects that are already developed by other developers. The architecture conforms to SOA as we illustrate in Figure 4. Indeed, object (or service) providers publish the functionalities they perform into a common registry, called “Interface Repository” in CORBA terminology. More precisely, they publish a description of their objects, a description which is written using the Interface Description Language (IDL). An example of an IDL file is depicted in Figure 5.

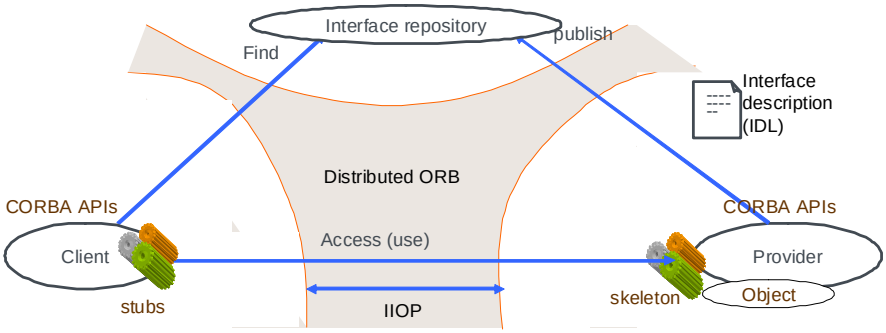


Figure 4: CORBA-SOA analogy.

```
interface Account {
        // Operations available on the account.
        void deposit(in float amount);
        void withdraw(in float amount);
        ...
};
```

Figure 5: IDL file example.

IDL describes the different operations that are provided by the object, their inputs, and their outputs. IDL files are programming language independent, but the service publication process, and service invocation process is performed using existing APIs (JAVA API, C++ API …). Nevertheless, thanks to Stubs and Skeletons, the client application remains completely independent from the server application. Stubs and Skeletons are mediation objects which are automatically created from the IDL files using CORBA IDL compiler, which are language-dependent. Still, the Object Request Broker (ORB) supports major programming languages. Stubs mediate between the Client and the ORB in order to transform the IDL method invoked by the client into an ORB operation, and Skeletons mediate between the ORB and the provider of an invoked object in order to transform ORB operation into an actual method invocation of the target object, which is programming language-dependent.

ORB component is responsible for routing a request to its target. This includes transmitting input to the target, waiting for the response, and transmitting the output to the requestor. It hides from the service consumer the real location of the invoked service. The Stubs and Skeletons ensure data marshal and unmarshal. This is necessary to ensure loose coupling between the service consumer and the service provider.

In order to support distribution over different entities, OMG has introduced IIOP; a specification of how to implement General Inter-ORB Protocol (GIOP) over the internet. GIOP is a specification that defines the formats of inter-ORB messages exchange. It enables two independent ORBs to communicate, and thus, a client from an ORB *A* can invoke a server object registered on another ORB *B.*

d. Web Services Architecture (WSA)

Web Services are currently the most popular technology that enables SOC. A Web Service [W3C, 2004b] is "*a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*" This definition highlights the different technologies used within Web Services Architecture (WSA): WSDL, SOAP, HTTP, and XML. Nevertheless, the architecture comprises also facilities for publishing and discovering services. UDDI (Universal Description, Discovery and Integration) is currently the common technology that provides such facilities, though other Peer-to-Peer (P2P) or hybrid approaches such as [Du, 2006], [Verma, 2005], and [Podesta, 2008] exist.

As we illustrate in Figure 6, Service providers create services, describe there interfaces using WSDL (Web Service Description Language), and publish them to a common registry using an UDDI

interface. Then, Service developers use a discovery interface to discover services within the UDDI registry, and invoke them by creating a SOAP (Simple Object Access Protocol) message.

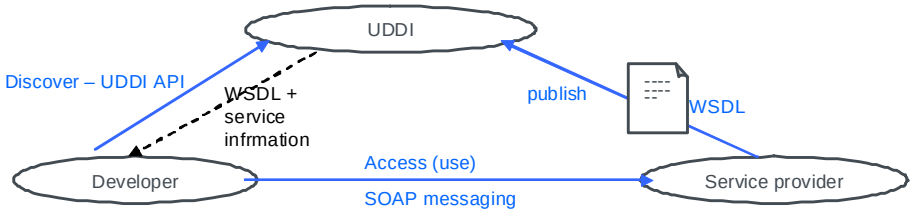


Figure 6: Web Service Architecture (WSA)-SOA analogy.

These standards (WSDL and SOAP) are oriented for a machine-to-machine communications. They provide a common way to describe, publish and invoke services. Figure 7 illustrates a WSDL file which describes a Weather service, and Figure 8 illustrates the corresponding SOAP request. Each WSDL file may describe one or several operations of a service. Each operation contains an abstract description and a concrete description. The former refers to the signature of the service and the latter defines how developers invoke it. The SOAP messages are then created accordingly.

```
<wsdl:types>
  <s:element name="GetWeatherByPlaceName">
    <s:complexType>
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="PlaceName" type="s:string"/>
      </s:sequence>
    </s:complexType>
  </s:element>
</wsdl:types>
<wsdl:message name="GetWeatherByPlaceNameSoapIn">
  <wsdl:part name="parameters" element="tns:GetWeatherByPlaceName"/>
</wsdl:message>
<wsdl:portType name="WeatherForecastSoap">
  <wsdl:operation name="GetWeatherByPlaceName">
      <wsdl:documentation>Get one week  weather forecast for a place name(USA)</wsdl:docum
      <wsdl:input message="tns:GetWeatherByPlaceNameSoapIn"/>
      <wsdl:output message="tns:GetWeatherByPlaceNameSoapOut"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="WeatherForecastSoap12" type="tns:WeatherForecastSoap">
  <wsdl:operation name="GetWeatherByPlaceName">
    <soap12:operation soapAction="http://www.webservicex.net/GetWeatherByPlaceName" style=
    ...
  </wsdl:operation>
</wsdl:binding>
```

Abstract Definition

Concrete Definition

Figure 7: Weather service description file.

```
POST /WeatherForecast.asmx HTTP/1.1
Host: www.webservicex.net
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.webservicex.net/GetWeatherByPlaceName"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ...
  <soap:Body>
    <GetWeatherByPlaceName xmlns="http://www.webservicex.net">
      <PlaceName>string</PlaceName>
    </GetWeatherByPlaceName>
  </soap:Body>
</soap:Envelope>
```

Figure 8: Weather service request and response.

Though WSDL-SOAP enable software features to automatically invoke a Web service (Create and send the SOAP request), the developer intelligence is still required to first choose which service and which operation respond to a specific need, and second to check whether the input parameters of the interface is semantically conform to the data that the software feature is willing to provide. In order to tackle this limitation, semantic technologies such ontology dictionary languages (Resource Description Framework (RDF) and Web Ontology Language (OWL)) [W3C, 2004e], semantic Web Service description languages (such as SA-WSDL and OWL-S), and semantic reasoning tools are associated to WSA [McIlraith, 2003]. This is known as Semantic Web Services (SWS). Figure 9 illustrates the new architecture. Service descriptions are semantically annotated using a common dictionary, and then, the platform provides semantic discovery tools.

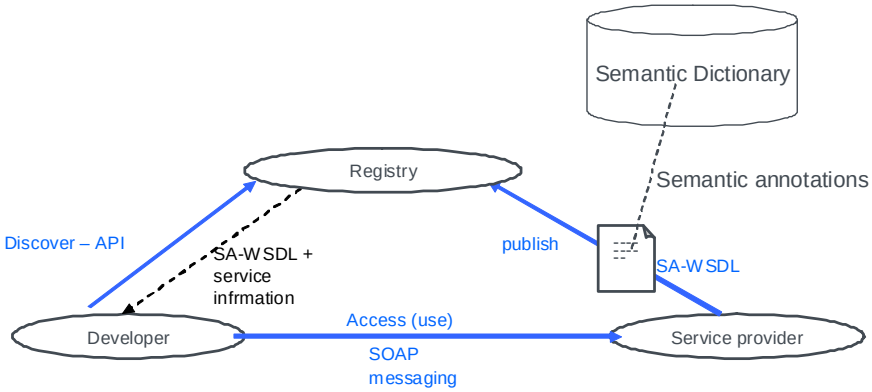


Figure 9: SWS architecture.

e. *REST*

REST architecture is coined by Roy Thomas Fielding in his Ph.D. dissertation [Fielding 2000]. It is characterized by representing each resource as a Uniform Resource Identifier (URI) accessible through HTTP. Thus, while WSA and CORBA are operation oriented, REST-based service architecture is resource oriented. Each resource is accessible through a unique URI with different methods (GET, POST, PUT, and DELETE). The GET method enables the requestor to read a resource. The POST method enables him to update a resource. The PUT method enables him to create a resource. Finally, the DELETE method enables him to delete a resource.

As an illustrative example, consider a contact list web service which implements three functionalities: get contact list functionality, get contact details functionality, and add contact functionality. In SOA, we expose these three functionalities as three operations, defined in the WSDL file, whereas in REST based service architecture, we expose the resources with URIs as follows:

- http://contactlist.org/contactlist/: this URI is accessible by GET and POST methods. GET method returns a document containing the whole entire contact list, and POST method adds a contact to the contact list.
- http://contactlist.org/contactlist/contactidentifier: this URI is accessible by a GET method and returns a document that contains the details of the contact.

REST service exposure method makes the service invocation as simple as making an HTTP request, so there is no need for a special API to invoke a service; almost all programming languages enable the creation of HTTP requests. However, as illustrated in Figure 10, REST in itself does not include a registry, which is necessary to publish and discover services. Therefore, Sun Microsystems published WADL, a Web Application Description Language, which is a WSDL-like description language format that aims to describe resources instead of operations. Thus, WADL files can be published to and discovered in the registry. More recently, even W3C added resource description support to the WSDL 2.0 specification.

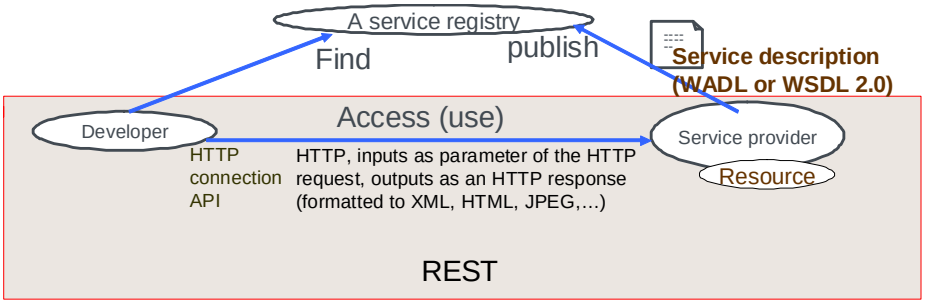


Figure 10: REST.

2.2 Service Composition using SOA

We classify SOA based service composition into three main categories: static service composition, automatic service composition, and semi-automatic service composition. Service composition is defined as the process of creating a composite service; which is a combination of services or capabilities that provides a new functionality. The following subsections detail each approach of composition and analyze the related technologies.

a. Static Service Composition in SOA

Static service composition is a combination of two or more services performed by a developer at the design time, using programming languages. This is the most basic approach to perform service reuse; the core concept of SOA paradigm. The term "static" indicates that the user can no longer modify it at runtime. WSA and REST-based architecture are currently the most used to enable the static service composition. From the technical perspective, this type of composition is carried out by providing to the developers APIs

that enable them to easily discover and invoke services. We distinguish two types of APIs: Backend APIs and Frontend APIs. This distinction is actually related to the programming language those API are intended for. The Backend APIs are thus intended for developers that use server side programming languages such as Java and PHP. We find for instance APIs such as Java SOAP client and PHP SOAP. The Frontend APIs are intended for frontend programming languages such as JavaScript, Java applets, and Adobe flash. We find for instance IBM Dojo toolkit extension and jQuery SOAP client.

In both cases, the APIs facilitate the creation of SOAP messages, and the retrieving of a specific parameter within a SOAP response of a Web Service. The most important advantage of this approach is the richness of the created application. As it is created by a developer, services can be deeply integrated with each other. In addition, the presentation layer of the application can be completely decoupled from the composition logic. This implies the possibility of creating sophisticated UI to enhance the user experience. However, the limitations of such approach consist in its difficulty, the long time to market it implies, and the difficulty to personalize the new application according to a specific user. First, it is difficult because the composition of services as well as their presentation to the user is performed using a programming language. Second, it implies a long time to market because the needs are first expressed by the user, and then processed by a developer. In addition, the request processing time includes the implementation of the business logic by calling the different services, and the implementation of the UI. Third, once the application is created and deployed, the user can not modify it. Each modification requires the intervention of a developer in order to reengineer the application.

b. Automatic Service Composition in SOA

The loose coupling between services in WSA and REST, and the availability of more and more basic services over the Web fostered the research on automatic service composition. Automatic service composition is characterized by creating automatically a composed service from a user request. Research work has been done extensively on semantic issues and natural language interpretation in order to build a customized service directly from a user request. As we illustrate in Figure 11, most approaches consider the SWS architecture (see Figure 9) and add two new components: User Request Interpretation and Automatic Service Composer.

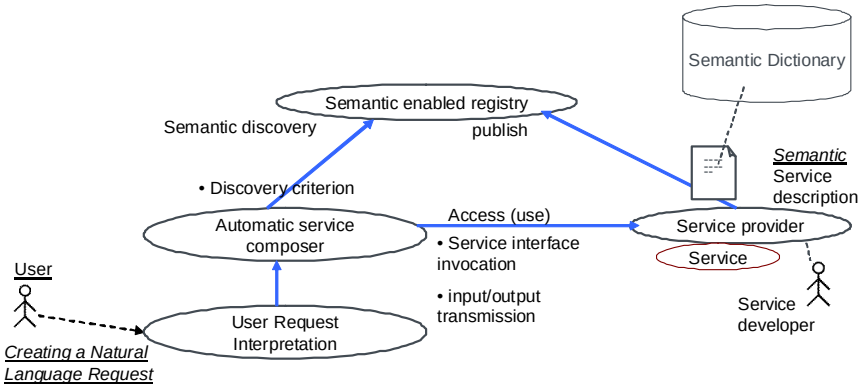


Figure 11: Architectural model for automatic service composition.

User request interpretation entity is in charge of transforming a natural language request to a formal request [Bharati, 1995]. It essentially detects the goals of the user and the available inputs.

Automatic Service Composer entity receives a formal request that contains user desired goals, with the available inputs, and it generates an orchestration of available services so that the user can reach his goals starting from the available inputs. There are two main approaches to do that: a top down approach, and bottom up approach. In the former, the algorithm starts from the goals, explores different orchestrations that reach these goals, and selects the one which matches the available inputs. And in the latter it starts from the available inputs, explores the available orchestrations, and selects those that reach the user's goals. To illustrate these two approaches we consider the work of Lecue and Leger [Lécué, 2006] and [Lécué, 2007]. Their algorithms are based on a causal link matrix (CLM [Lécué, 2006]) or the extended CLM (CLM+ [Lécué, 2007]). While CLM is a matrix that represents all matching possibilities between inputs and outputs of services, the extended one takes into account the non-functional properties. Computing the CLM matrix includes the quantification of the similarity of two concepts in the semantic database (the ontology). This quantification is based on the logical relations between the concepts. Table 2 shows an example of such quantification.

Table 2. Quantification of semantic matching of parameters.

Logic meaning	Signification	Value
$S1 \equiv S2$	Semantic of S1 is exactly the same as semantic of S2 according to an ontology Θ	1
$S1 \leq S2$	S1 is a subclass of S2	2/3
$S1 \geq S2$	S2 is a subclass of S1	1/3
$S1 \neq S2$	S1 is different from S2	0

Figure 12 illustrates an example of a CLM matrix. Lines refer to all entries parameters of all services and columns refer to all inputs of services and to the goals of the user request. An element in the CLM is a set of vectors $V\ (l,\ c) = (S_i,\ Value)$ where S_i is a service that has as input l, and *Value* is a semantic matching value between an output of S_i and the corresponding column parameter c (which is an input of another service).

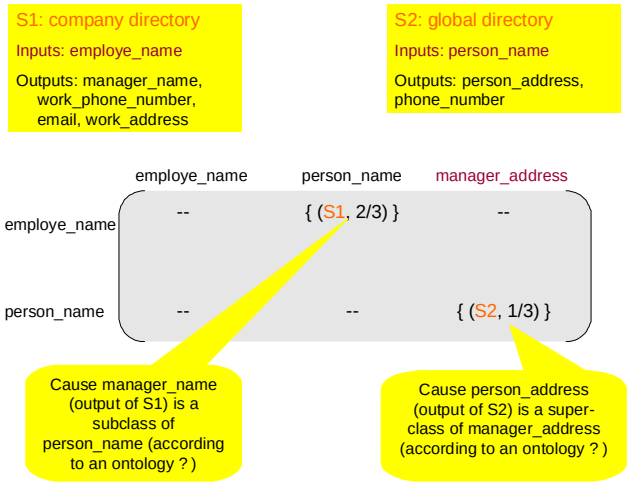


Figure 12: CLM simple example.

Pa4C, which belongs to a bottom up algorithms, is a recursive algorithm that runs on the top of the CLM. It has as input the constructed CLM, a set of available web services (WS), initial user inputs that represent the initial knowledge base (KB), and the user goals (B). The algorithm then populates the KB with reached parameters through available services in WS set until all user goals are reached.

Graph based algorithms follow however the inversed reasoning approach (the top down). Such as the Pa4C algorithm, we have as inputs: the constructed CLM, a set of available web-services (WS), the user goals (B), and the initial knowledge base (KB). A set of services N is initialized with services that have as outputs user goals B. For each service (S) in N, the algorithm checks if user inputs include all required parameters for its execution. If this is the case, service S is removed from the list and the algorithm proceeds to next one until N is empty. If user inputs are not sufficient to allow the execution of the service S, then the algorithms checks in the CLM+ if there are services that provide as outputs the necessary parameters. If such services are found then we remove S from the list and we add the found services to N. The composite service is constructed as the algorithm populates N.

Automatic service composition is definitely an ideal approach for composing services as it enables users to get dynamically a new service responding to their spontaneous needs, expressed using their natural language. However, it still suffers from research issues that prevent it to reach a potential industrialization. Examples of such issues are the natural language ambiguity [Cremene, 2009] and how to

provide a comprehensive dictionary of concepts that enable the description of all existing services [Pop, 2009]. These two issues give rise to the inaccuracy of the Automatic Composition (The generated composite service does not match exactly the user needs). Furthermore, the UI part of the service created automatically using this approach is very basic. This can be explained by the absence of the UI layer in WSA.

c. Semi-automatic Service Composition in SOA

Semi-automatic service composition category includes any mechanism that enables the user to define a composite service with more investment than making a natural language request. The idea is to let developers to create basic services and users to intuitively chain them in order to create more innovative ones. Figure 13 shows the basic architectural model of this category of service composition. We first need to create the basic services and publish them into a registry. Then, users discover the needed services in the registry and compose them through an "Orchestration definition graphical tool". This will generate a script that defines the composition logic. Finally, the created composite service can be executed using an orchestrator (also known as execution engine), which uses the already created script that defines the composite service.

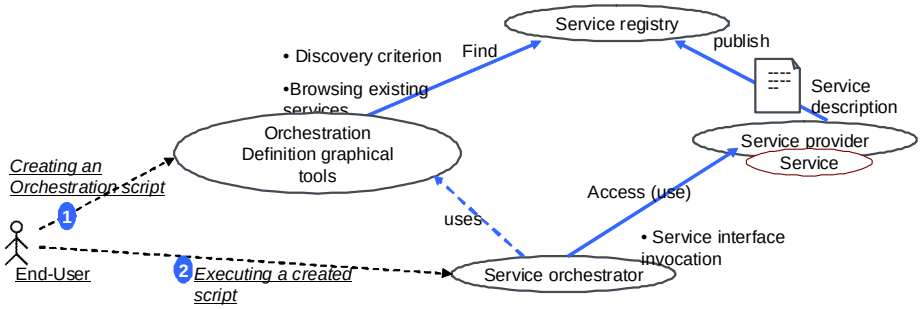


Figure 13: Semi-automatic service composition general model.

This model is essentially fostered by the current emerging concepts of Web 2.0, where software applications are fragmented into web services and users are actively involved in content creation process. Indeed, content in websites like Wikipedia, Youtube, and Flikr[22] is essentially populated and maintained by the users themselves.

After an investigation of existing tools, we have classified them into three main sub-categories:

- graph-based service composition,
- desktop environments based service composition,
- and websites-based service composition.

[22] Flickr, http://www.flickr.com/, visited on January 15th, 2010

In the following parts of this section we will details these categories.

Initially, graph-based service composition tools were addressed for developers and advanced users in order to speed-up the service creation process. Several frameworks such as JBPM [Cumberlidge, 2007], Eclipse BPEL designer[23], [Ku, 1994], and jABC (Java Application Building Center) [Margaria, 2006] [Steffen, 2007] have emerged. These frameworks are all based on process definition languages such as BPEL (Business Process Execution Language) [Andrews, 2003], JPDL (JBPM Process Definition Language) [Cumberlidge, 2007], and SLG (Service Logic Graphs) in [Ku, 1994] and [Steffen, 2007]. The creation tools are thus based on intuitive graphical interface that enable the assembling of black boxes according to the logic of the service needed by the user. The logic is defined by chaining the black boxes by mapping outputs and inputs. Other operations like conditions and loops could be added graphically as well. The black boxes are referred through different names depending on the community. But, they are all conceptually equivalents; they are reusable software features, with well defined functions and interfaces (inputs and outputs). Thus, in JBPM we use the term *task* to refer to an activity realized by a human or a software (business process community), and in jABC (telecom community) we use SIBs (Service Independent Building Blocks) to refer to a reusable software feature.

Though these tools definitely make the creation process easier and faster, they still remain addressed for developers and advanced users. First, they are based on IDEs (Integrated Development Environment) which are hardly accessible by ordinary users. Second, it is necessary to understand different computing concepts such as (flowchart, inputs, and outputs) to be able to create new services.

Other frameworks, that follow exactly the same model depicted in Figure 13, attempted to get closer to the user by implementing the composition capability directly at the Web browser level, either as a Web application or as a browser plugin. However, they still remain based on defining a flowchart based on different and sometimes complex (for ordinary user) operations such as regular expressions, web services invocation, conditions, loops…etc; concepts that are not understood by ordinary users. Consequently, we claim in this thesis that this type of composition is addressed for advanced users. Nevertheless, as the composition logic is defined independently of the user interface, this type of composition provides a significant flexibility in the UI creation. They enable developers and advanced users to create sophisticated UIs. Examples of such frameworks are: Yahoo PIPES, MARMITE [Wong, 2007], and OPENMASHUP[24]. Figure 14 shows an example of a composite service in Yahoo PIPES. Boxes represent basic services and wires represent input/output mapping between these services. In that example, we have used three services:

[23] Eclipse BPEL designer, http://www.eclipse.org/bpel/, accessed on July 31, 2010
[24] Open Mashup, http://www.openmashup.org/, accessed on July 31, 2010

String builder service (let the user to enter the input), translation service, and Yahoo search service. Wires link the output of string builder service (String) with the input of the translation service (text), and the output of the translation service (text) with the input of the Yahoo search service (String). As a result, we have created a new service which translates a string passed as input, and search on the web for the translated string using Yahoo search engine.

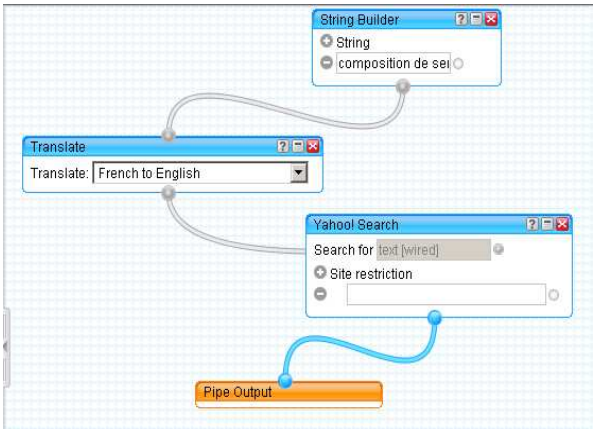


Figure 14: Yahoo PIPES screenshot.

MARMITE and OPENMASHUP follow exactly the same philosophy as Yahoo PIPES. Services are represented through boxes and user should set up the mappings between outputs of services with inputs of others. This approach however is suited only for advanced users; users that know what an input and an output of a service are and how to chain them. This assertion is supported by experimentations that have been conducted in [Wong, 2007] as well as in our own experimentations detailed in *Chapter III.3 Experimentation and* . Authors in [Wong, 2007] have tested their framework on a sample of six persons: two of them are experienced in software programming, two others are experienced in spreadsheet but not programming, and the others are not experienced in programming nor in spreadsheet. The results have shown clearly that this type of composition is addressed for advanced users. Indeed, only three out of six have succeeded to build a composite service. More important is that, those who have succeeded are those who have knowledge in programming and spreadsheet. This experience demonstrates that this type of composition framework is more addressed to advanced users than ordinary users without any computing technology knowledge.

The second category of semi-automatic composition tools is based on the desktop environment of the user. It includes mechanisms that involve the user in the creation and execution of composite services within their desktop environment. he history of this type of composition have begun with traditional desktop environments such as Microsoft Windows, MAC operating systems, and Linux that enable the users to compose two independent desktop applications using intuitive mechanisms such as OLE clipboard,

more known under the name of *copy and paste*, and OLE drag & drop. However, these mechanisms still have three limitations:

- They do not consider current Internet of services where users are no longer limited to local applications. Users use more and more remote web services such as Map, online search, and online purchase.
- They suffer from the late failure detection. In other words, the system does not detect compatible applications for a copied or dragged data, and propose them automatically to the user. Instead, the destination application of the paste or drop action is responsible for retrieving the compatible data (data that it can handle) from the clipboard.
- The created composition is not rich as it is limited to data moving (and optionally to format transformation) from an application to another.

In order to tackle these limitations, the emerging solutions firstly include a desktop environment that considers Web-based services, and secondly define composition tools that enable the users to combine services within their desktop environment. Examples of such approaches are EZWEB [Soriano, 2006] and IBM Mashup Center[25]. Both require user participation to make the composition, which makes them belonging to semi-automatic service composition. Figure 15 illustrate the corresponding model, which is different from the traditional semi-automatic service composition illustrated in Figure 13. Service providers create widgets and publish them into the catalogue of the framework. Users can browse that catalogue, select the desired services, and load them into their environment. Thereafter, users can launch a "Wiring" interface, which enables them to define sequences of widgets execution. These wires are then executed as the user uses his environment.

Figure 16 illustrates how do users create a link between two services in EZWEB. It illustrates a YouTube search service linked to a YouTube player service. This link is created by the user himself using a dedicated user interface that enables the selection of an output of a service and mapping it to an input of another. This Figure also shows the execution of the created wire.

[25] IBM Mashup Center, http://www-01.ibm.com/software/info/mashup-center/, accessed on July 31, 2010

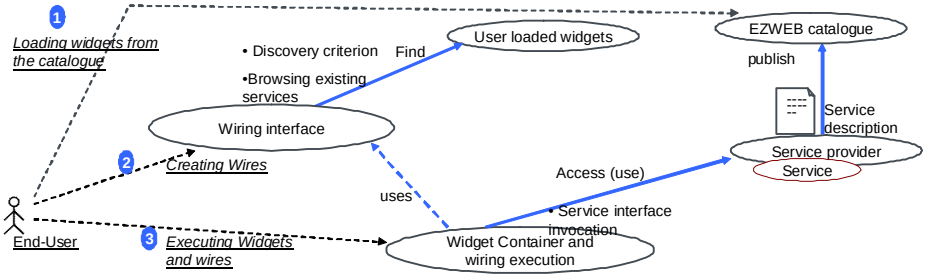


Figure 15: Semi-automatic composition model in EZWEB.

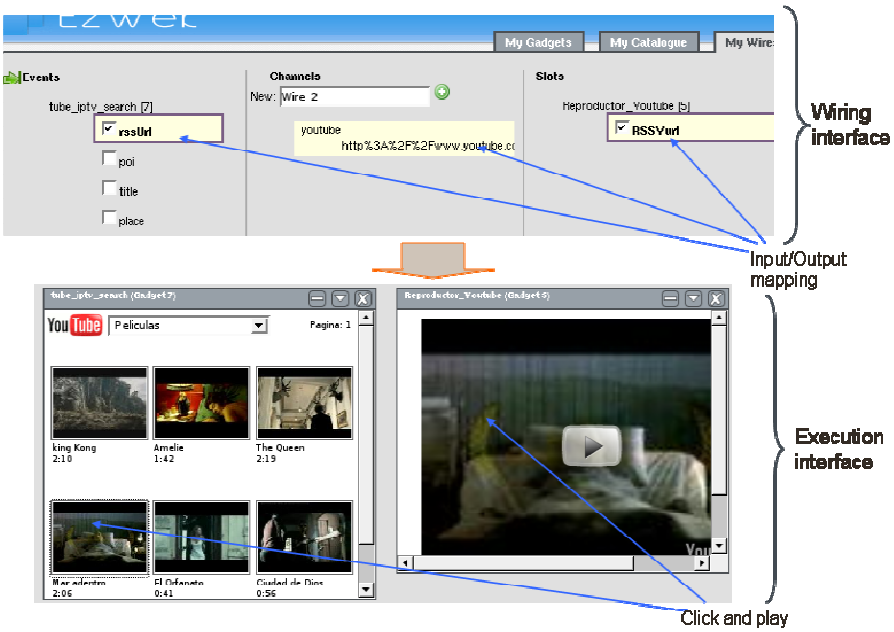


Figure 16: EZWEB screenshot.

The last identified category of semi-automatic service composition is based on Web-sites. It includes mechanisms that enable users to add functionalities to already existing websites. This does not modify the website itself; instead the mechanism modifies the current instance of the website (the response generated for a user request). MASHMAKER [Ennals, 2007a] and [Ennals, 2007b] and MARGMASH [Díaz, 2007] are two examples of such mechanism. MASHMAKER is implemented as a Firefox plug-in. Its most important innovation is the extraction of semi-structured data from web pages. Consequently, users can map that data to inputs of existing services. Figure 17 for instance displays a "Yellowpages" web page in which MASHMAKER component has extracted automatically all addresses, phones and names. Thereafter, the user has mapped the addresses into a Map service. To do that, he has just loaded the Yahoo Map widget.

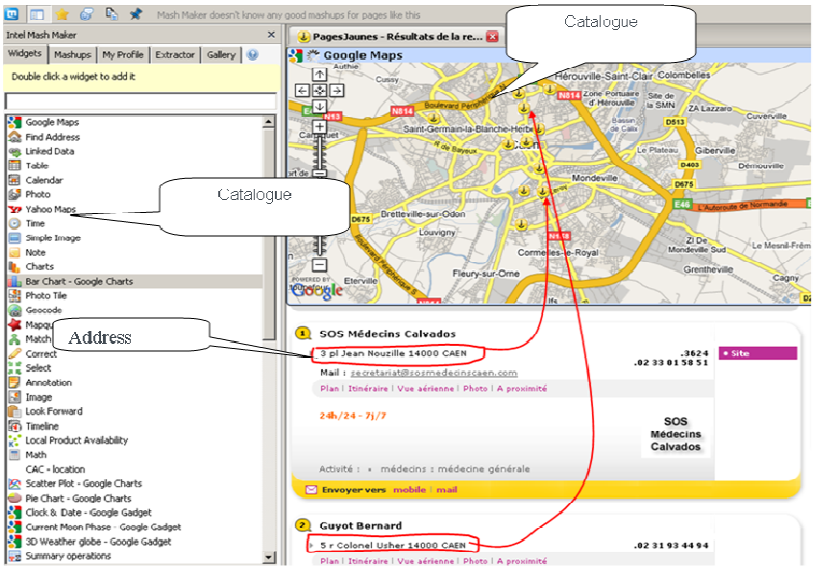


Figure 17: MASHMAKER screenshot.

These tools enhance significantly the intuitiveness of the composition framework as the composition script is created automatically as the user loads widgets to the website. However, the created service can not be sophisticated because all wires must go from the Web site outputs to widgets inputs. Thus, all created composite services follow the schema illustrated in Figure 18.

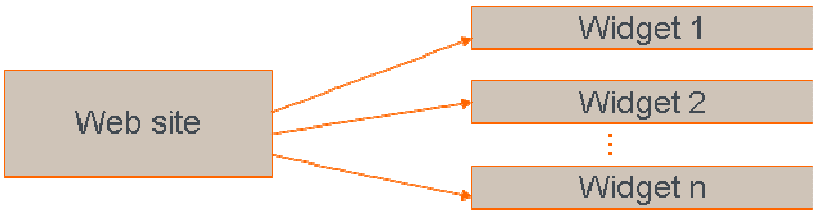


Figure 18: Composite service schema.

d. Comparison

Table 3 depicts a comparison of the different composition approaches according to identified criteria. It takes into account both user and service developer point of view:

- Coupling between services: This criterion characterizes the dependencies between two services that a service composition approach requires.
- Service developer investment: This criterion characterizes the additional investment, required from a developer who wants to make his service “reusable”.
- Personalization: User needs are various, numerous, and evolving. Therefore, personalization capability is an important characteristic for an efficient composition tool. This criterion characterizes the ability of the user to personalize a created service to his needs.

- Distribution over user devices: This criterion characterizes the capability of creating composite services which are distributed over the user devices; so that each atomic functionality will run in an appropriate device.
- Richness of the created service: This item reflects what kind of and how sophisticated are the applications that can be created by developers and users, using a composition approach? It reflects also the richness of the UI of the composite service.
- Intuitiveness of the composition tool at the design time: The intuitiveness of composition tools is an important requirement to attract more users and developers. Especially in the context of Web 2.0 where we expect ordinary users, without development skills, to compose their own services.
- Time to market: How to reduce the time to market of applications is a question that rises in most companies. The TTM starts from the instant where the user needs a new service and ends when he gets it. TTM includes the service request processing time, the development time, and the deployment time. While static service composition tools aim to reduce essentially the development time, automatic and semi-automatic aim to reduce all these phases.
- User needs matching: It is important to create services that match exactly the user needs.

Table 3. Comparison of service composition categories.

	Automatic	Semi-Automatic			Static
		Graph-based	Desktop environments based	Websites-based	
Coupling between services	High	Medium	Medium	Medium	Low
Personalization	High	Medium	Medium	Medium	Low
Distribution over user devices	No	No	No	No	Yes
Richness of the created service	Medium	Medium	Medium	Low	High
Intuitiveness of the composition tool at the design time	High	Low	Low	Low	Very low
Time to market	Low	Medium	Medium	Medium	High
User needs matching	Low	High	High	High	Medium +

The most important advantages of automatic composition of services are: personalization, and the time to market of new services. Automatic composition of services is very centred on the user; the service creator is the user himself and the creation time is almost the execution time. This means that the user can

modify a created service easily and instantly, which makes the automatic composition of services very customizable. Also, the automatic approach reduces significantly the time to market as the service is created directly and automatically by the user. However, this composition category suffers from some limitations:

- Tight coupling between several entities: automatic service composition requires a common semantic vocabulary between several entities: service platform, different service providers, and even the users when making their request (they should use the same vocabulary in their request as the one defined in the service platform).
- The composite service can not be sophisticated as a developer application: Because it is done automatically, and it is a composition of building blocs, the created service can not be as sophisticated as a service which is created directly by a developer from scratch. In addition, the UI these tools generate for a composite is rudimentary.
- The composite services are not distributed over the user devices.
- Difficulties to match exactly the user request: due to current limitations of semantic reasoning and natural language processing, the created composite service does not always match the user request. This imposes a validation step (performed by the user) before the execution. This limitation is also due to the difficulty for the user to express their request in an efficient way (in a vocabulary close to the platform's one).

In static service composition, the most important advantages are: the loose coupling between services, the richness of the created service, and the user needs matching. Services are independent from each others because developers do not really need semantic tools to understand the functionalities of a service. The created service can be as sophisticated as the developer can do, there are no particular technical limits as the service invocation is done by the developer using programming languages. However, this category of composition mechanisms suffers from the incapacity of personalization, and time to market. It is not customizable for two reasons: the first one is that the creation time is different from the execution time, and second, users need development skills to modify the execution sequence of services. In addition, the time to market is very high as the entity which needs the service is not usually the entity which develops the service. Therefore, it requires a development request processing phase and a development phase.

The last composition category is semi-automatic. We have categorized it into: Graph-based service composition, desktop environment based composition, and websites based service composition. Semi-automatic service composition represents a tradeoff between the static and automatic service

composition in term of several criteria such as time to market, loose coupling between services, intuitiveness of the composition tool, and personalization. It also excels in users needs matching criterion compared to automatic and static composition as in semi-automatic composition the users create themselves the composite service, which matches exactly their needs. In addition, they can modify it whenever they want to.

2.3 Business Process Management using SOA

There are many definitions of the term business process: [Hammer, 1993], [ebXML, 2001], [Aguilar-Savén, 2004], and [Ko, 2009]. However, an invariant seems to rise among all these definitions. Indeed, all these definition mention that a business process is a set of coordinated activities performed to reach a business goal.

Business Process Management (BPM) is the action of discovering, modelling, developing, executing, and monitoring business processes that are pertinent in a given organization. It aims to provide organizations with efficiency in performing their business activities, while ensuring their agility – the capability of quickly adapt and modify their business processes.

In this section we review the approaches and technologies that are used to manage business processes, and then, we figure out the advantages and limitations of current practices.

a. Discovery and modelling

The task of discovering and modelling business processes is currently performed by business analysts. The discovery usually follows either a top-down approach or a bottom-up approach [Verner, 2004]. The top-down approach is characterized by identifying and defining at first the high level business processes, and then, decomposing them into lower level processes until the lowest, user or profile specific, level is reached. This approach has the advantage of providing a high level organization insight of the processes, and the limitation of lacking in process details and accuracy [Verner, 2004].

The bottom-up approach, in contrast to the top-down approach, starts from capturing the business processes of the lowest, user or profile specific, level. Then, the processes are generalized and/or merged each other to form end-to-end processes of an organization [Verner, 2004]. This approach has the advantage of being detailed and accurate, but business analysts can hardly have a broad picture of business processes of an organization.

In both approaches, business process discovery also suffers from finding the best trade-off between the accuracy of the business processes and their number. Indeed, the more business analysts try to enhance the accuracy of business processes, the more these processes are specific to a limited number of

users, which augments their number and complicates their implementation and maintenance. As an illustrative example, let us consider a vacation request business process. The high level view can be summarized in three activities: vacation request creation, send the vacation request to the requestor manager, study and making decision, notifying the result to the requestor. However, as we try to get more precise in the actions that are performed by users, additional versions of the process appear. Figure 19 for instance, illustrates two precise processes of a team manager and purchasing and logistic responsible. After receiving a positive response to a vacation request, the team manager updates his agenda, sends email to his team, and sets up an automatic email response, while the purchasing and logistic responsible searches pending purchasing orders, calls the providers, redirects the incoming calls during the vacation period, and updates his agenda.

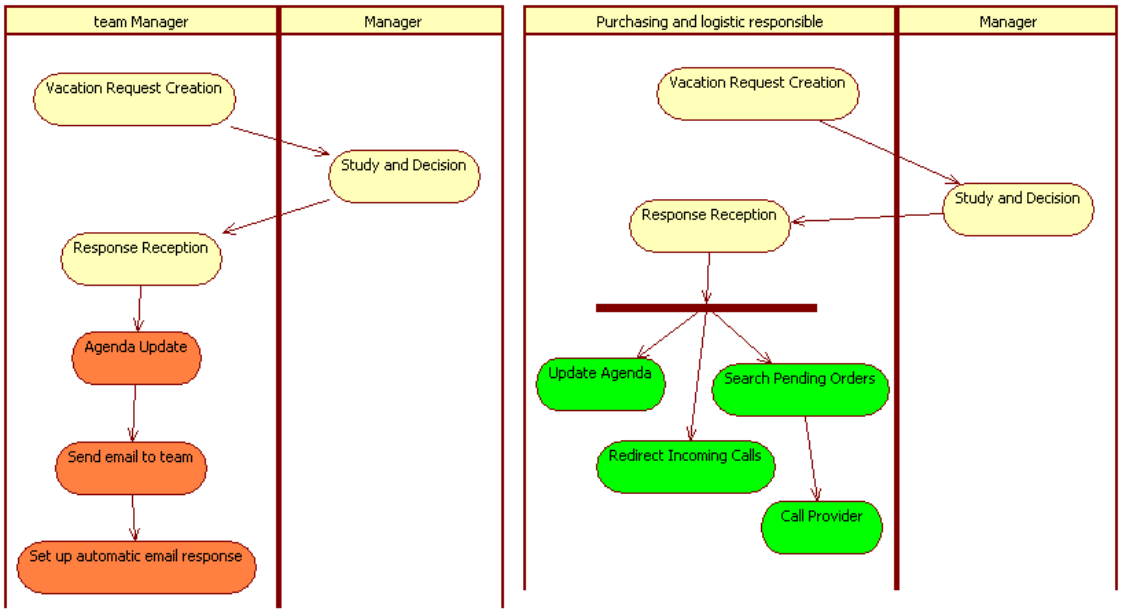


Figure 19: Vacation request business process versions.

After discovering an organization business process, business analysts should model it in order to share it with other business analysts, as well as with developers to automate it using a software application. The most common way to model business processes is using BPMN (Business Process Modelling Notation) [OMG, 2008b]. It is a flowchart diagram very similar to activity diagram of UML (Unified Modeling Language). BPMN is an OMG (Object Management Group) standard whose last version is released on 2008. The most important challenge in the modelling of business processes is filling the gap that exists between business analysts and IT teams. It is required to represent the details that developers need.

b. Development and execution

The most important challenge in the development of business process management applications is how to foster agility in the organization. WSA, associated to composition languages, are currently the most promising approach that embraces this challenge. It is characterized by exposing business enterprise applications as Web Services, and using service composition languages to combine these services to implement a business process. The developer most important task is to associate a Web Service to a business activity defined in the BPMN model. Figure 20 summarizes the WSA approach for implementing business processes [Arsanjani, 2007].

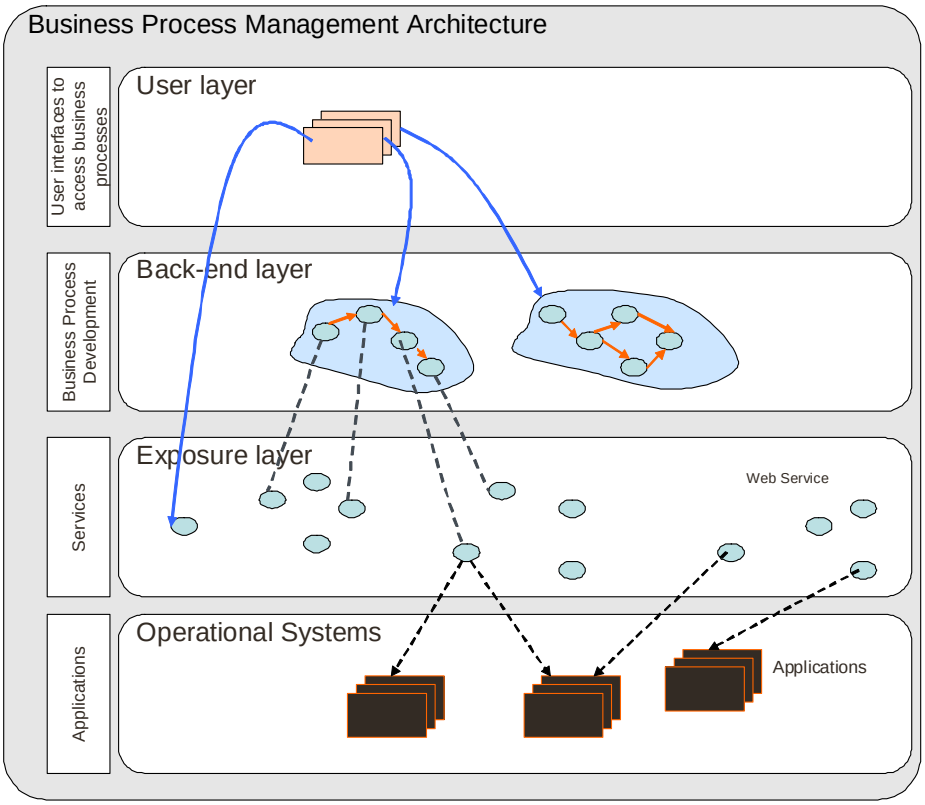


Figure 20: Business Process Development using WSA.

As we illustrate in Figure 20, the development of business processes are completely independent from the applications that are running within the organization. Functionalities of such applications are exposed through Web Services (WSDL/SOAP), and developers can invoke and compose these services based on a business process definition, without any modification on the applications. The composition of those services is either performed manually, using SOAP APIs, or semi-automatically using scripting (XML-based) languages such as BPEL4WS, which are much easier. In addition, these languages are usually empowered with graphical tools (e.g. Eclipse BPEL editor) that facilitate even more the composition process. Figure 21 for instance illustrates a BPEL4WS graphical representation of a

composition of services made using Eclipse BPEL editor. It shows a News service combined to a send Email service.

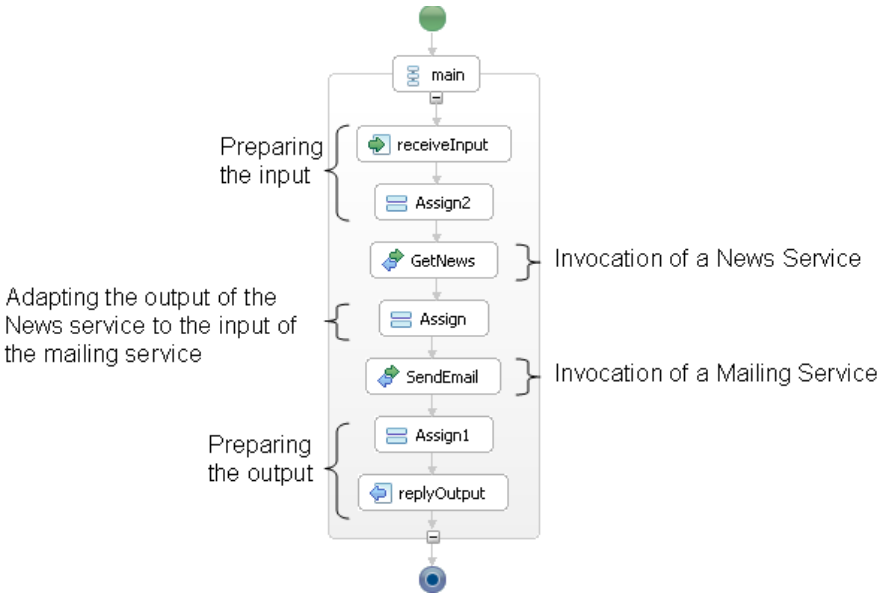


Figure 21: BPEL4WS Graphical Representation.

c. End-to-end Sequence diagrams

Figure 22 shows respectively the end-to-end process of creating and updating a business process. The first step is the discovery a relevant business process. This can be a request from the user as illustrated on the Figure, or a proactive action performed by the business analyst. After capturing the needs, the business analyst models the business process and requests a development of an application. The developer then translates the model into an executable BPEL script and creates the application. At this step, the end-user can access and use the application. When a specific update is needed by the user or captured by the business analyst, it is required to change the model, and send it to the developer who updates the BPEL script as well as the UI of the application if needed.

d. Advantages and limitations

Though BPEL4WS associated to WSA provides a significant flexibility in the development of business processes, it still has some limitations:

- First, the end-to-end process of developing business processes still remains long and difficult to perform. It is long, because it includes the process discovery step, the process modelling step, and the process development step. It is difficult, because of the difficulty of capturing specific user needs at the right time.

- Second, the loop between the users, the business analysts, and the developers in the process of updating a business process makes the adaptation to spontaneous users needs impossible.
- Third, the implementation of a business process, which is characterized mainly by the definition of a BEPL4WS script, is tightly coupled to the invoked Web Services. This implies updating the BPEL4WS script each time a change on the Web Service that performs a given business activity occurs.

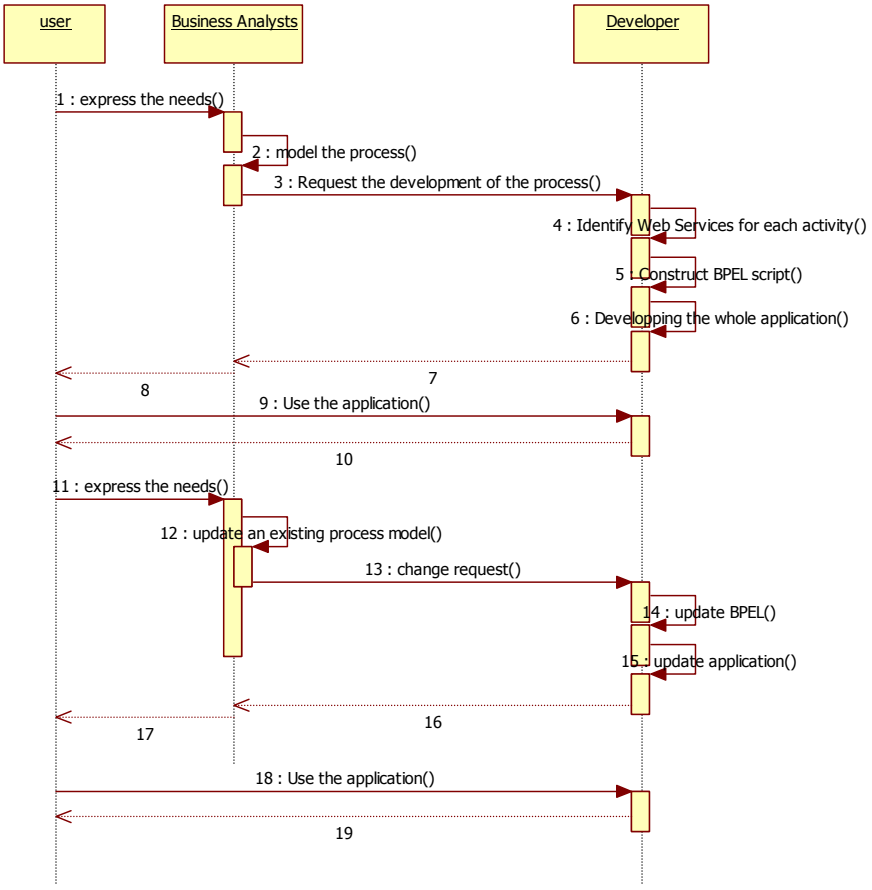


Figure 22: End-to-end sequence diagram of business process modeling and development.

2.4 Conclusions

WSA and REST-based architecture are currently the two main alternatives that enable SOA, and consequently enable SOC. As far as it is used by developers, they address perfectly the need of integrating different services with each other to create a new one. However, our survey shows clearly that these technologies are not addressed at all for users. They do not enable ordinary users to compose services mainly due to the technology complexity. As Figure 23 illustrates, services are usually described and

invoked using XML-based format such as WSDL, WADL, and SOAP; technologies which are not understandable by ordinary users at all.

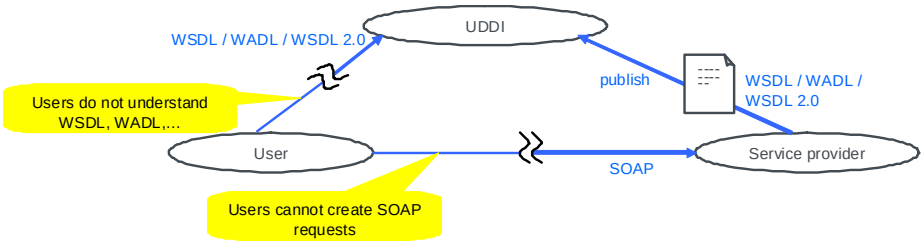


Figure 23: Technology gap between users and WSA (and REST).

Mainly due to this technology gap between what is used and what is understandable by ordinary users, several limitations are perceived in service composition and business process management fields. Table 4 illustrates these limitations. It also summarizes the advantages of SOA.

Table 4. Current SOA advantages and limitations.

Item		Advantages	Limitations
Service composition	Static service composition	• Services are completely independent from each other. The created service might be as sophisticated as an ordinary application (No technical limitation). • Enable the creation of distributed applications. • The created service matches exactly the user needs.	• The creation process is complex. It is conceived only for developers. As a consequence, a long TTM for personalizing an existing service, as well as for creating a new service is noticed. • The created service is tightly coupled to the used basic services.
	Semi-automatic service composition	• Addressed for advanced users. It enables personalization. • The time to market (TTM) is low when a user is able to create a service (He is an advanced user). • The created service matches exactly the user needs.	• Not designed for ordinary users; the tools are too complex for them. This implies a long TTM when an ordinary user wants to create a new service, or to personalize an existing one. • The created service is limited. • The created services can not be distributed over the user devices. • The created service is tightly coupled to the used basic services
	Automatic service	• Designed for ordinary users. • It enables a quick creation of a service.	• The created service can hardly match exactly the user needs. • The created services cannot be

	composition	• It is very intuitive.	distributed over the user devices. • The services are tightly coupled as they rely on a common semantic. • The created service is tightly coupled to the used basic services
Business process management		• SOA enables a seamless integration of enterprise business processes. It hides the implementation aspects of enterprise applications. • Graphical tools such as BPEL significantly speed up the development of business processes (but it still performed by developers).	• Business processes are heterogeneous, and thus it is hard to capture and implement all the details. In other words, business processes are generalized for the sake of simplicity. • Adaptation to new processes is long as it requires first the capturing of the need; and second, its development (usually by a different entity). • The business process integrator is tightly coupled to the Web Service they use. • Unstructured data are not captured by business process integrators

3 Service Environments

The machine-to-machine interaction has been massively investigated within SOC. But the human-to-machine interaction is not. We believe that this is the main reason that makes SOA fails in user generated services, and the related fields such as service composition and business process management. In this section, we review the technologies that enable software to get closer to the user. We provide more details on Web portals technology, as it is the current trend. Nevertheless, we summarize the evolution history to this technology (Web portals).

The most common user service environment is the operating system of the user machine (such as Linux and Microsoft Windows). Their goal is twofold: abstracting the use and the management of hardware components, and providing the necessary infrastructure that enable users to intuitively manage and run their applications. In this thesis we essentially focus on the second item and classify in this regard these environments into three models.

3.1 Model 1

The first model covers both initial versions of desktop environments such as MS-DOS and UNIX, as well as more recent ones such as Microsoft Windows and Linux. It consists either in a command-line interface (CLI) or WIMP (Windows, Icons, Menus, and Pointer) interface. Figure 24 summarizes the model.

Figure 24: Model 1 overview.

Before getting a service running, users should first procure and install on their devices both the service environment (e.g. UNIX or Microsoft Windows) and the desired services. Thus, at the runtime, the user device, the provider of the user environment, as well as service providers are completely independent. However, the user is limited to run the service at that specific device; the device in which the user environment and the service is installed. This presents a significant constraint especially to highly mobile users.

3.2 Model 2

The first model has been the adopted for many years. But the emergence of Web 2.0 and the increasing number of services available on the Web have encouraged industrials to think about new models. Model 2 and 3 are examples of Web 2.0 aware service environments.

Model 2 actually refers to the approach adopted by Google in its Google Chrome operating system. It is characterized by installing at first the user environment at the user device, and then users can dynamically, and on demand, access and use remote services, which are mostly running at the service provider platform as illustrated in Figure 25.

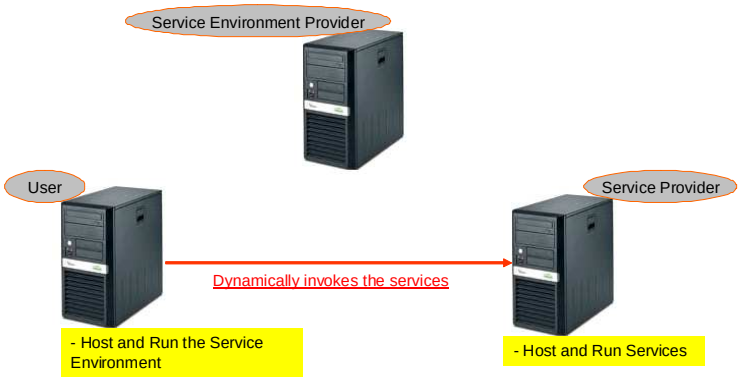


Figure 25: Model 2 overview.

The most important advantage of this approach is the adoption of the software as a service (SaaS) paradigm [Armbrust, 2009]; the users can dynamically (on-demand) invoke software services hosted and running remotely on the Web. However, it requires an installation of the service environment on the user device, which, such as model 1, presents a constraint to the highly mobile users.

3.3 Model 3

Model 3 is another example of "Web 2.0-aware" service environments. It is characterized by entirely adopting the SaaS paradigm; both software and service environments are considered as services that are hosted and running remotely on the Web. This is also known as DaaS (Desktop as a Service) [Beaty, 2009]. Figure 26 summarizes this approach.

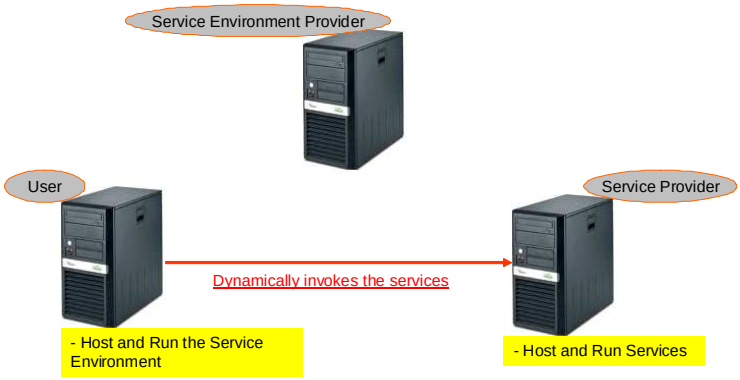


Figure 26: Model 3 overview.

This model includes several frameworks that we classify into two main categories: Web operating systems (OS) category ([Weiss, 2005] and [Lawton, 2008]) such as Wiki-OS[26], Glide OS[27], and eyeOS[28], and Customizable Web Portals category such as [Bellas, 2004], MyServices[29], iGoogle[30], and Netvibes[31].

[26] Wiki-OS, https://www.wiki-os.org/
[27] Glide-OS, https://desktop.glidesociety.com
[28] eyeOS, http://www.eyeos.com/
[29] Orange MyServices, http://www.espace-utilisateur.orange-business.com/index.php, accessed on July 31, 2010

The former tends to produce the same environment as traditional operating systems [Lawton, 2008] (managing the same type of applications that package several functionalities), except that the managed applications are hosted in the Web. Whereas the latter, in addition to managing services that are hosted in the web, it promotes personalization by enabling users to create their own environment by loading only functionalities they need. These functionalities are wrapped within small UIs called Widgets [W3C, 2007] or Portlets.

Each category has its advantages and limitations. As illustrated in Figure 27.a, in Web OSs, functionalities of an application are tightly coupled by a developer according to pre-requested users' needs (e.g. Microsoft Outlook contact list functionality is coupled to send email functionality). However, one limitation of this approach is that the application is hardly customizable by the user; if users need a new functionality to be integrated to an existing application, they must express their need to a developer who is in charge of integrating the functionality. Furthermore, the communication between applications is limited to what was expected by the developer during the development phase. In other words, the developer of application A could call an application B (or a web service), if, and only if, during the development of application A, the developer knows about the availability of application B at the runtime.

In Customizable Web Portals however, applications are split into a set of widgets in the same way as we split applications to Web Services in SOC; each widget gives access to an independent functionality. This method enables users to customize their environments by loading only the needed functionalities. But the limitation of such environment stems from the fact that the developed functionalities are independent from each others. Users can not launch a send email service from a contact list service, or a "Google Map service" from a directory search results. Figure 27.b summarizes the characteristics of this category.

Customizable Web Portals are characterized by aggregating small user interfaces called Widgets [W3C, 2007], Portlets [Sun, 2003] [Sun, 2008], or Gadgets. In the next section, we briefly describe the differences between these concepts, and then we overview the OASIS initiative to standardize the relationship between Web Services and Portlets.

---

[30] iGoogle, http://www.google.com/ig?hl=en
[31] Netvibes, http://www.netvibes.com

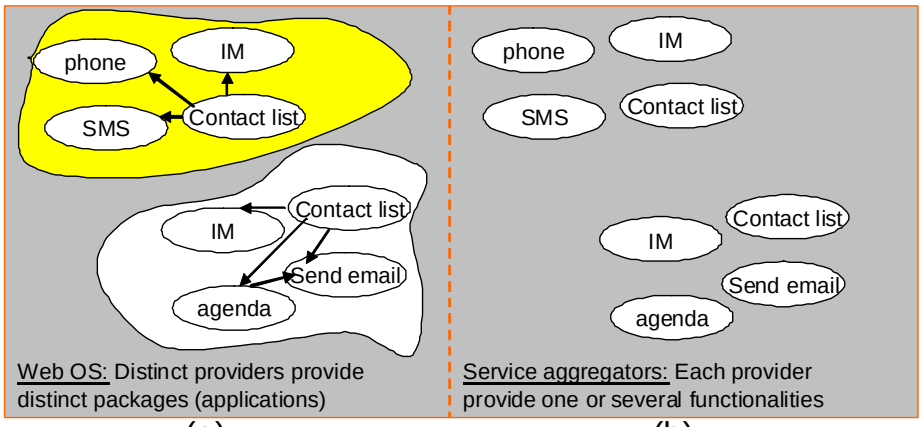


Figure 27: Comparison of Web OSs and Customizable portals.

4 Widgets Related Concepts

Though there are several implementations of the Widget and Widget aggregation concept, we can distinguish two main approaches: the Java community with JSR 168/286 specifications, and the Web community with W3C and UWA specification. Before merging us to the details of those specifications, we first show in Figure 28 the common characteristics between the two approaches. It is illustrated that the model is very simple: each Widget has an implementation and a description. The implementation is characterized by different modes, which are described and referenced in the description of the Widget. The modes include basically: a View mode, a Configuration (or edit) mode, and a Help mode. The View mode corresponds to the main screen of the Widget that enables the user to interact with the different functionalities provided by the Widget. The Configuration mode corresponds to a screen that enables the user to customize the Widget. It enables him to set configure the Widget according to his preferences and specific parameters. These parameters are then saved and reused in further access to the Widget. Finally, the Help mode provide users with documentation on the usage of the Widget. While the help and the Configuration modes are optional, the View mode is mandatory as it provides the core logic of Widgets.

In addition to these basic modes, a Widget aggregator may define additional modes that Widgets should or must provide. Typically, it is interesting to integrate a notification mode to Widget aggregators where communication Widgets (e.g. Telephony, IM, and email) might be integrated. By notification we refer for example to incoming calls, new message…etc. This provides a unified notification zone for different services (e.g. at the bottom-left of the screen). The additional modes provide an additional constraint to Widget developers, as they should implement additional UI fragments within the UI of their functionalities. But, they also provide a deeper integration between different Widgets, and thus enhance the user experience.

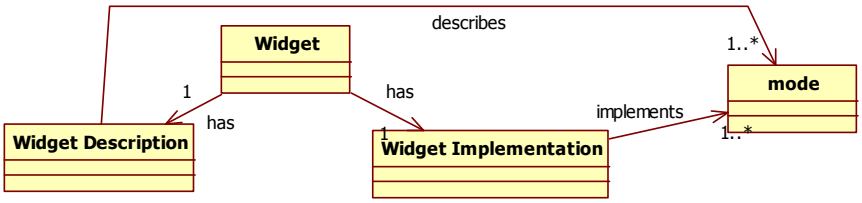


Figure 28: Comparison of Web OSs and Customizable portals.

The following subsections details the different technologies and how this model is realized.

4.1 Portlets (JSR 168/286)

Java Portlet Specification [Sun, 2003] defines a Portlet as "*a Java technology based web component, managed by a portlet container, that processes requests and generates dynamic content. Portlets are used by portals as pluggable user interface components that provide a presentation layer to Information Systems*". This definition highlights the main characteristics of a Portlets. First, it is a Java technology. Second, it is used by portals as pluggable UI; UI that can be added and removed from the portal by users or portal administrators. Third, it needs a portlet container as an execution environment, and a portal as a front-end presentation environment. Figure 29 depicts a high level overview of interactions between Web Portals, Portlet container, and Portlets.

Portlets are initially specified in JSR 168 [Sun, 2003] – also known as *Java Portlet Specification 1.0*. This specification aims to standardize the interactions between Portlet Containers and Portlet providers. The goal is to achieve interoperability so that each Portlet would be able to run on all Portlets container complying with the specifications. JSR 286 [Stefan, 2008] (V2.0) is the second version of the Portlet specification. It aims to overcome the limitations of 5-years experience of the first version.

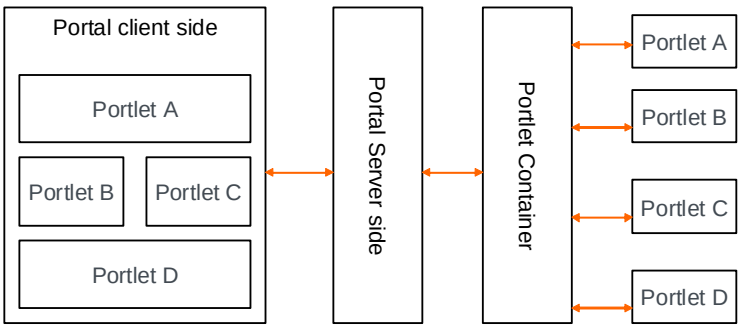


Figure 29: Portlet High Level View.

From the technical perspective, a Portlet is a Java Web application that follows a specific API. This API is characterized by a Java interface that must be implemented by the Portlet. This interface

contains a set of functions whose most important are described in Table 5. We illustrate also a typical scenario that involves calls to these functions in Figure 30.

When an action (e.g. submitting a form) is performed on one Portlet (e.g. Portlet B), the portal invokes the Portlet container. Then, the portlet container:

- First, invokes the *processAction* of the corresponding Portlet (B).
- Second, invokes the *processEvent* function of each Portlet which has subscribed to an event generated by Portlet B, following the *processAction* function.
- Third, invokes the *render* function of each Portlet – this function generates the UI fragment that will be displayed on the portal.
- Fourth, transmits each fragment to the Portal, which is in charge of generating the new Web Page.

This scenario illustrates the limitations of JSR 286. Indeed, in addition of being limited to Java technology, JSR 286:

- Does not really make use of AJAX (Asynchronous JavaScript And XML) capability of Web browsers. Though it enables a Portlet to use AJAX to invoke its server side logic, the Portal itself can not use AJAX to update a single Portlet.
- Portlets are not completely independent from each other. Indeed, each time an action is performed on one Portlet, the Portlet Container invokes the *render* function of all other Portlets and updates their UI. This implies a significant constraint for developers as they can hardly manage different states in the UI level.

Table 5. Portlet Interface Description [Sun, 2003].

Function Name	**Function Description**
Init()	This function is calls by the Portlet container immediately when a new instance has been created. It can be used by the Portlet developer to initialize his variables.
processAction()	This is the main function that interacts with the user. Developers can add links or forms that users can fill and send back to the Portlet *processAction* method. Here, developers can implement the business logic of their application by invoking Web Services, or querying the data base,…etc.
render()	This function is called immediately after the *processAction* function. It aims to enable the developers to generate the UI fragments. The *processAction* and *render* functions together enable developers to separate the business logic implantation from the presentation layer. The *render* function is executed each time a Portlet is updated in the Portal.

processEvent()	JSR 286 specification enables Portlets to communicate events between them. This requires from the Portlet Developers to first subscribe to an event type (in the Portlet configuration file); and second implement the *processEvent* function, which is executed each time such event is generated by other Portlets or the Portlet container.
doView, doEdit, doHelp	This corresponds to the different modes of the Portlet. According to the Portlet state (View, Edit, or Help), the *render* function calls one of these functions (*doView, doEdit, doHelp*). If for instance the portlet is on the state *Edit*, the render function calls the *doEdit* function in which developers should prepare the HTML form to enable users to personalize the Portlet.
destroy()	This function is executed when a Portlet instance is about to be destroyed by the Portlet container

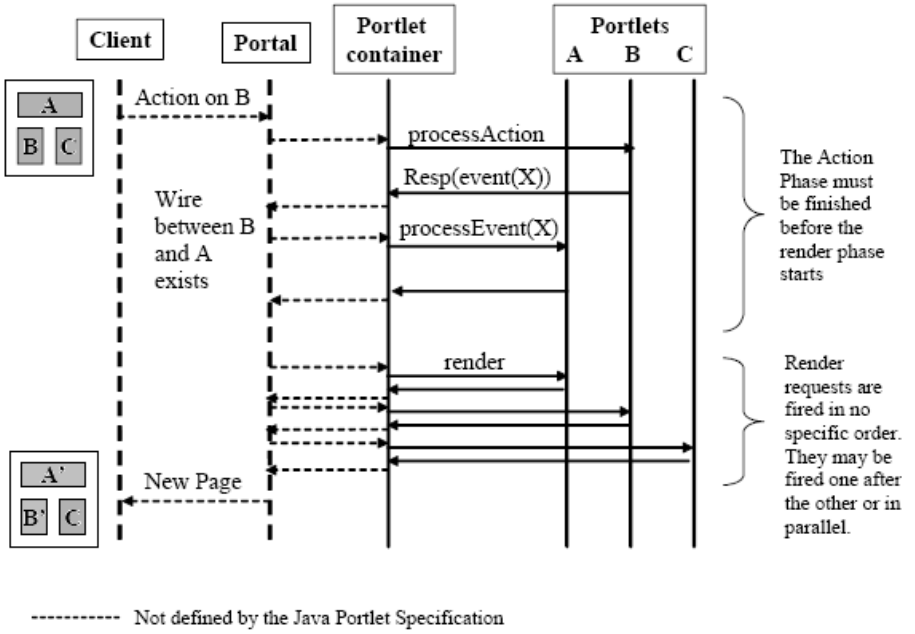


Figure 30: Request Handling Sequence ([Stefan, 2008]).

4.2 Widgets

From the user perspective, a Widget is exactly the same as a Portlet: It is a pluggable UI component, used by Web Portals and provides a presentation layer to Information Systems. Unlike JSR 286 which is specific to J2EE platform, Widgets do not impose any Server side technology. Instead, it must be a Web application, which means that the client side of the application must be based on Web technologies such as (X) HTML, JavaScript (JS), CSS, and XML. W3C for instance defines a Widget [W3C, 2007] as "*a self-contained client side Web application for displaying and updating remote data, packaged in a way to allow a single download and installation on a client machine or mobile device.*" Universal Widget API (UWA)

[UWA, 2008] is another initiative, launched by Netvibes, to perform interoperability between different Portals. It promises a single development and several deployment platforms paradigm. In other words, developers create Widgets using the UWA specification, and then, the Widgets can be automatically adapted to a specific Widget engine. Currently, all major Widgets engines (e.g. Netvibes, iGoogle, Windows Vista, Apple Dashboard, Yahoo! Widget, iPhone, Opera, blogs, MySpace, etc.) support the UWA format.

Unlike JSR 168/286 which mainly standardizes the interactions between the server side of the Portlet and the Portlet Container, W3C and UWA focus on the interactions between Portals and the client side of the Widgets. They specify how to define a Widget and what technologies to use for implementing the client side. UWA for instance, requires XHTML as the language for presenting the content of the Widget, XML for handling the preferences, JavaScript for adding behaviour to the Widget, UTF-8 as the encoding format, and finally, CSS for styling the Widget. In addition, UWA compliant Widget engines provide a set of JavaScript functions that aim from one hand to provide facilities to developers (e.g. AJAX requests), and from another hand to limit the developer to ensure that Widgets do not interfere with each other. Table 6 summarizes the most important JS functions.

Table 6. UWA JavaScript Functions.

UWA Function	**Description**
widget.onLoad()	This function should be rewritten by the Widget Developer. Its content is executed when the Widget is loaded on the Portal.
widget.onRefresh()	This function should be rewritten by the Widget Developer. Its content is executed when the Widget is refreshed.
widget.setAutoRefresh(delay)	Developers can use this function to automatically refresh the Widget each *delay* minutes
widget.setTitle(title)	Changes the title of the Widget.
widget.openURL(url)	Opens a new URL on a new browser window
widget.body	Provides a reference to the Widget root element.
widget.createElement	Creates a DOM element.
widget.getValue(name)/ widget.setValue(name, value)	Respectively gets and sets values of preferences parameters.
UWA.Data.request(url, request object) (Alternatives: getFeed, getJson, getXml, and getText)	Performs an AJAX request
addContent, appendText, setText, setHTML, setStyle	Modify the content or the style of an element.

By focusing on the client side aspects, W3C and UWA provide a significant flexibility for developers in the implementation of the business logic from one hand, and for portal developers in the implementation of their portal from another hand. Indeed, in JSR 168/286, both Portlet developer and Portal developer must use Java technologies and rely on the JSR 168/286 specification in the implementation of their server side logic. These constraints explain the growing popularity of Widgets compared to Portlets. Indeed, Widgets do not impose any constraint in the development of the server side logic, for both Widget developer and Portal provider.

4.3 SOA vs. Portlets and Widgets

SOA is a mature paradigm widely used among software developers. It is empowered with several technologies which enable them to seamlessly integrate ready-to-use (third-party) services within their software (Service or application). However, current technologies such as Web Services and REST architecture, which enable SOA, still lack of presentation layer. They still lack of the UI that enable the service to interact with the user and vice versa. On the other hand, Widgets and Portlets are by definition pluggable UI fragment that respectively provide access to data and business logic of an application. But, current Widget technologies did not deeply investigate the interoperability and integration aspects between Widgets. In addition, the relationship between Widgets and SOA architecture is not deeply investigated.

First attempts to converge SOA and Widget/Portlet paradigm have been made by OASIS Standardization Group, in the Web Service for Remote Portlets (WSRP) Specification ([OASIS, 2003] and [OASIS, 2008]). WSRP aims to provide Web Service developers with an easy way to embed a pluggable UI. This enables WSRP compliant Portals to easily integrate a Web Service. As we illustrate in Figure 31, the standard is characterized by adding a common interface to WSDL description files. This common interface returns the UI of the Web Service. This UI enables the user to interact with the service and vice versa. Portal providers from the other hand should implement a generic Portlet that should invoke the WSDL common interface which returns the UI, and render the results to the Portal.

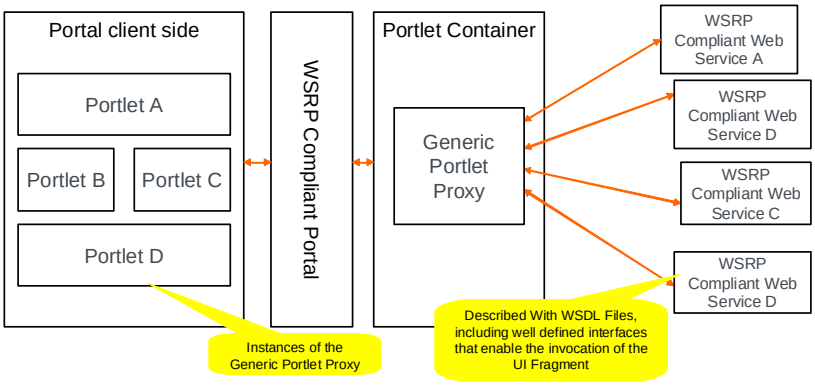
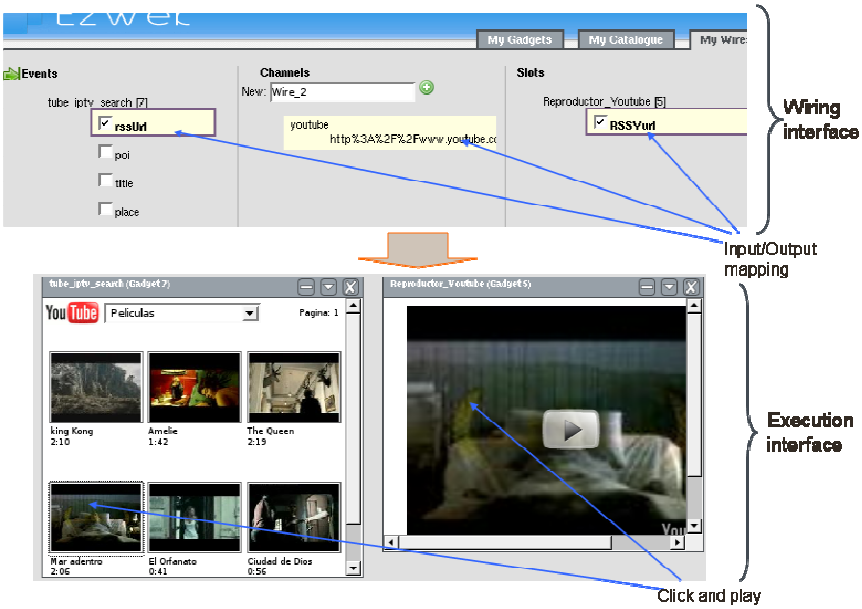


Figure 31: WSRP Basic Concepts.

Though WSRP standard is limited to JSR 168/286 compliant Portals, the idea of adding a face to Web Services is pertinent. As supported by [Díaz, 2008] and [Akram, 2005], this idea of adding a user interface to Web Services makes Portlets in Web Portals playing the same role as Web Services in SOA, namely enablers for application assembly. We witness even researchers advocating development methods and the necessity of orchestration tools based on Widgets/Portlets ([Díaz, 2008], [Vo, 2006], [Sire, 2009], and [Soriano, 2006]). In [Sire, 2009], authors proposed an API that enables developers to develop Widgets which communicate with each other when they are loaded in the same portal. In [Soriano, 2006], authors have proposed EzWeb platform; a Widget aggregator that enables Widgets to communicate with each others through wires defined by the users. The EzWeb platform is illustrated in Figure 16, which is duplicated hereafter.



Current solutions are not deeply investigated and compared to traditional SOA regarding different domains. In this thesis, we define a comprehensive architecture that enables users and developers to combine Widgets with each other at the Web portal level. Then, we apply this architecture to two SOA application fields, namely service composition and business process management. We aim to to highlight where SOA and Widgets technologies complement each other and where they do not.

5 Semantic Related Technologies

SOA might be empowered with semantic tools in order to facilitate the composition of services (static, automatic, or semi-automatic). In this section, we summarize the current practises in adding semantic to a service, whatever its nature (Widget, Web service, Web application…etc.).

5.1 The Different Approaches of Semantic in the Web

Such as services, semantic has a producer and a consumer. While the consumer could be a human or a machine, the producer of semantic is inevitably a human. This is due to the definition[32] of semantic itself; *an adjective of a relating to the meanings of words*. The *meaning* can not be generated by machines.

The most common approach to add semantic to a software service is to describe it using a natural language. This is already used in software engineering when developers create documentation of their software, or when adding comments to their functions, in order to be understood by third party developers. It is also used in WSDL files in the <documentation> tag. However, such description is limited to humans. Indeed, the semantic consumer, as well as the semantic producer, are inevitably humans and not machines. As a consequence, only static service composition is supported. Automatic and semi-automatic, in which machines should build, and/or check the feasibility of, input/output mappings, are not supported.

In order to enable machines to consume semantic (understand the meaning of words), developers need a new approach for defining their services; this is known as semantic Web, coined by Tim Berners-Lee in [Berners-Lee, 1998]. For this purpose, "machine oriented" semantic technologies have emerged. We distinguish two popular approaches: the ontology approach and the markup languages approach.

a. Ontology Approach

The term ontology has been firstly used in philosophy field. It means *a theory of the nature of existence*. In computer science field, the term is used to refer to a formal model of knowledge in a specific domain. This formal model is characterized by a set of concepts and relationships between them. It is used in artificial intelligence to model real objects and enable automatic reasoning on their properties.

From the technical perspectives, there are many languages that enable the specification of domain ontology. Resource Description Framework (RDF) [W3C, 2004d] is an example of such language. This language is characterized by a set of triplets *T (Subject, Predicate, Object)*. This form of triplets enables the representation of a relationship (predicate; also called properties) between two concepts (Subject and Object). Thus, a concept is defined with a tag and its relationship with other concepts. An RDF model can be represented as a graph, where nodes refer to the different Subjects and Objects referred in the triplets, and arrows represent the predicate (relationship). The RDF specification defines a set of properties (subClassOf, typeOf, domain, range…etc.), but others could be added. Figure 32 is a simple example of an RDF model.

---

[32] Collins, http://www.collinslanguage.com/results.aspx, accessed on June 14th, 2010

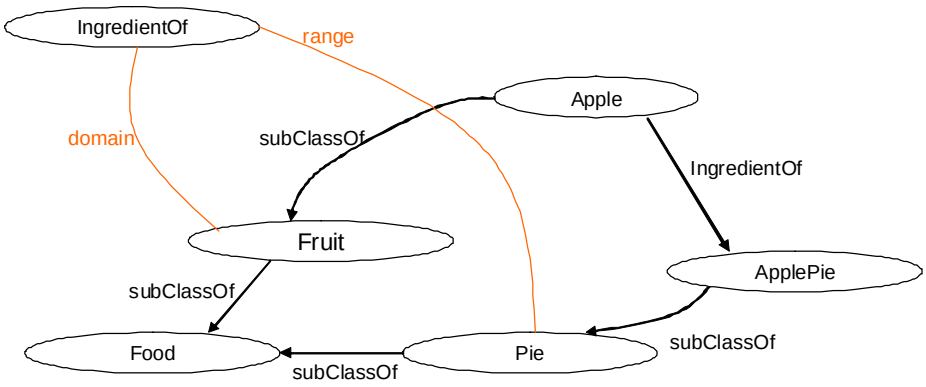


Figure 32: RDF graph example.

Another popular language that enables modelling ontologies is OWL [McGuinness, 2004], which stands for Web Ontology Language. It is an RDF based language, which provides more native predicates to enable reasoning. It has three sublanguages: OWL lite, OWL DL, and OWL Full. OWL Lite is designed for easy implementation. It restricts the language constructs to basic RDF schema such as: class, subClassOf, property, and relations between that classes and instances such as: equality and cardinality. In OWL-DL, developers can express value range of parameters, union, intersections and complement; these additional properties are those supported by description logic. Finally, OWL Full uses exactly the same constructs as OWL DL. The differences reside in the fact that OWL DL imposes some restrictions which are not imposed in OWL Full. An example of such restrictions is that in OWL DL classes, data types, data type properties, object properties, annotation properties, ontology properties are separated [W3C, 2004c]. This means for example that a class can not be an instance or a property, which is possible in OWL Full and RDF. With the imposed restrictions, OWL-DL ensures the existence of a decidable reasoning procedure with current reasoning tools.

b. Markup Languages Approach

The markup languages approach for adding semantic to software features is characterized by defining standardized tag, with clear and approved semantic (meaning), to be used by developers to annotate the information their application generates. It is mainly used in the Web, with, or in conjunction with (X)HTML.

The most known and successful technology to add semantic to (X)HTML documents is the *microformats* initiative[33] [Khare, 2006]. It is characterized by the definition of a set of formats to represent information used in Web applications. Examples of such information are: addresses, phone numbers,

[33] Microformats, http://microformats.org/, accessed on June 16th, 2010.

contact cards, calendar events, and email addresses. Table 7 is a summary of two popular specifications of microformats: hCard and hCalendar.

Table 7. Microformats examples.

Microformat tag	Sub-elements tags (Summary)	Description

hCard	- fn
	- adr (type, value)
	  - post-office-box
	  - extended-address
	  - street-address
	  - locality
	  - region
	  - postal-code
	  - country-name
	- email (type, value)
	- tel (type, value)
		Inspired by RFC2426, the hCard microformat aims to provide a representation of contact cards of persons, companies, organizations, and places. The main properties of this microformats are the name (*fn*), the postal address (*adr*), the email address (*email*), and the phoning address (*tel*). Note that for each address, we can specify its type. The type can refer for example to "work" for a professional address, or "home" for a personal address.

hCalendar	- dtstart
	- dtend
	- summary
	- location
	- attendee
	- geo
	  - latitude
	  - longitude
		Inspired by RFC2445, the hCalendar aims to provide a representation of calendar events. It contains for that purpose mainly the starting date of the event (*dtstart*), the ending date (*dtend*), a summary (*summary*), the location (*location*, *geo*), and one or several attendees (*attendee*).

These tags are used directly within the (X)HTML document, usually within *class* of *rel* attributes. Consequently, third party developers can easily request the Web page and extract useful information.

5.2 The Different Expressiveness Degrees of Semantic

In term of expressiveness, we can easily see that microformats do not provide the same level as ontologies created using RDF or OWL. For instance, we can not model a complete domain of knowledge (e.g. biology, wines, painting…etc) using microformats, as it is possible using ontologies. As a consequence of this lack of expressiveness of microformats, reasoners may detect two semantic matching between that are actually not compatible in the considered domain. For example, let's consider two services, one of them expects as inputs postal addresses located in France, and another generates US addresses as outputs. When using microformat, it is impossible to highlight the above difference in the model, as both addresses are referred through the "adr" tag. But this is possible using ontologies.

However, more we try to be more expressive in an ontology, more the ontology is unstable, and more the reasoning tools are heavy.

5.3 Semantic and Service Creation

As we have previously detailed, there are different approaches that enable different actors to create services through composition: static, automatic, and semi-automatic. The Web semantic tools provide support essentially for the automatic and semi-automatic composition approaches. In automatic composition approaches, semantic tools enable the detection of semantic matching between outputs of services with inputs of others. Pa4C algorithm for example, used in [Lécué, 2006], uses a semantic reasoning tool that enables detecting if two concepts (typically an output of a service and an input of another) are semantically close or not. In other words, the semantic reasoner detects if one concept can be substituted from another.(if they are equivalent (usually linked with properties like subClassOf, or Is), or if one is a sub element of another (usually linked with properties like UnionOf, or subElementOf)). Figure 33 shows an example where we use semantic substitution to compose services. In this example, semantic reasoning tools enable the composition framework to detect that S1 can be composed with S2 (in other words, we can substitute a phone number information from an IT Engineer information).

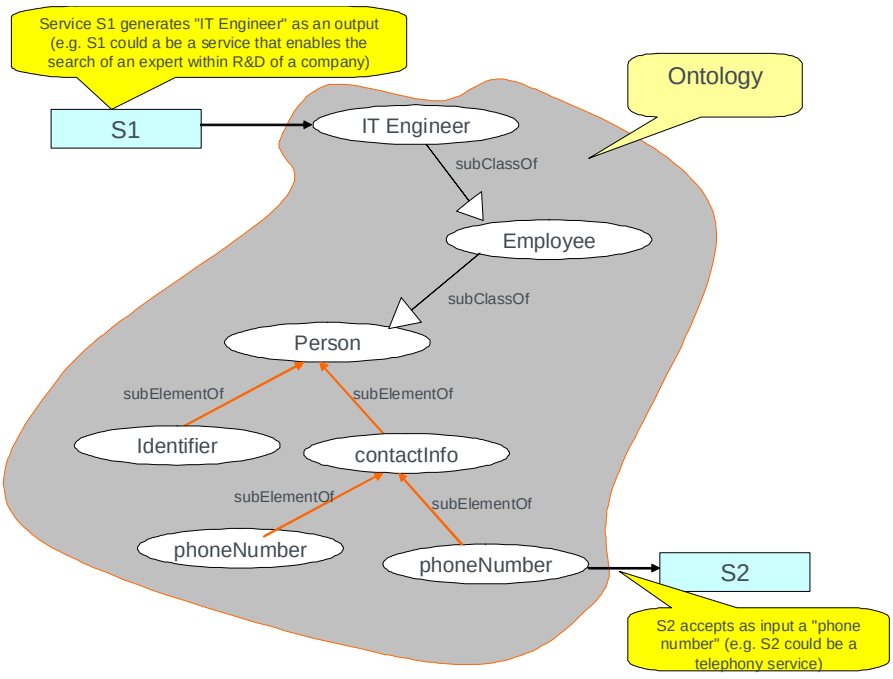


Figure 33: Example of semantic concept substitution in service composition.

In semi-automatic composition, Web semantic technologies are not mandatory. Indeed, as the task of mapping outputs and inputs of services is performed manually, by humans, it is not necessary to add

semantic technologies. However, they still remain useful to first recommend pertinent services in the composition process; and second, to check the validity of created composite services. These two actions are even more important when the composer of the service does not have computing skills. The former enhances the intuitiveness of the composition process, and the latter, ensures the semantic validity of the new service.

While in automatic composition the semantic must be highly precise and expressive, in semi-automatic composition, we can afford lightweight semantic as the composition process involves human intelligence.

6 Conclusions

In this Chapter, we have studied two paradigms: the Service-Oriented Architecture (SOA) paradigm and the Widget paradigm. Our main conclusion is that the former is not user-centric and designed mainly for developers, and the latter is addressed for users but does not consider the service (Widget) reuse (one main principle of SOA paradigm).

The SOA is not user centric as it is based on technologies that are understandable only by developers (e.g. XML, SOAP, and WSDL). As a consequence, user service creation is not really supported. Besides, two additional limitations are observed in current SOA:

- First, the reusability is based on well structured and formatted inputs and outputs of services. The unstructured data such as the exchanged addresses in an IM discussion are not considered. Consequently, much data are unavoidably discarded by current tools.
- Composite services are tightly coupled to the basic services they use. The unavailability of one used basic service implies the non-validity of the composite services that use it.

These limitations are not actually limited to service composition field. They impact also other fields such as business process management.

By contrast, the Widget paradigm presents the advantage of being designed for users. From the conceptual perspective, Widgets presents significant similarities with Web services in Web Service Architecture (WSA), and REST resources in REST-based architecture. Both enable exposing application functionalities; both are self-contained; and both are self-describing. The Widget aggregator is a software application that enables the user to build a personalized service environment, where different Widgets of different providers are displayed within the same window (e.g. Web page). However, the reuse capability (one main principle of SOA) is not really investigated within the Widget paradigm.

Our study of the two paradigms (SOA and Widget) shows that each one has its advantages and limitations. This raises the challenge of constructing an architecture that includes the best of each field,

namely the reusability capability of SOA and the user-centricity of Widgets. In the next Chapter, we seek to propose to enrich the Widget paradigm in order to build such architecture; we name it the Widget-Oriented Architecture (WOA) paradigm.

Part II Contributions

Our contribution in this thesis can be summarized in three items:

- First, we introduce the principles of the Widget-Oriented Architecture (WOA) paradigm.
- Second, we design the framework that realizes the principles of WOA paradigm.
- Third, we study and propose the application of the WOA paradigm to two SOA application fields, namely service composition and business process management.

Thus, this contribution part is divided into three Chapters that detail respectively the three contributions.

Chapter II.1 Widget-Oriented Architecture (WOA) Paradigm

As we illustrated in the state of the art Chapter (*Chapter I.1 State of the Art*), SOA still suffers from being developer-centric and not user-centric. In this Chapter, we introduce a Widget-Oriented Architecture (WOA); a new user-centric paradigm that aims to overcome current SOA limitations by providing the capability to ordinary users to create services based on the Widgets. Thus, we start by summarizing our vision about the definition of a Service; then, we specify the different roles involved in the new architecture; and finally, we detail the principles that must/should be fulfilled by each role.

1 Service

In this thesis, we define a Service as "*a software entity that performs one or more operations. It is developed by a service developer and has at least one service description. It is made available by a service provider and* ***consumed by another software entity or by a user****, who is optionally charged for*." Thus, unlike enablers which are designed to be used by other software entities, and unlike Applications which are designed to be used directly by the users, a Service in this thesis should be designed as a component which could be used by other software entities or by users.

These conceptual differences between Enabler, Application, and Software Service imply significant technical choices summarized in Figure 34. Thus, such as Web Services in traditional SOA, an Enabler (Figure 34.a) is technically described by a description file following a machine readable language such as WSDL and WADL. It is implemented using any programming language such as Java and PHP, and can be invoked by third parties through well defined transport and messaging protocol such as HTTP and HTTP/SOAP. Because they are designed to be programmatically processed, the presentation aspects are not useful in an Enabler, except if it is part of the Enabler business logic itself.

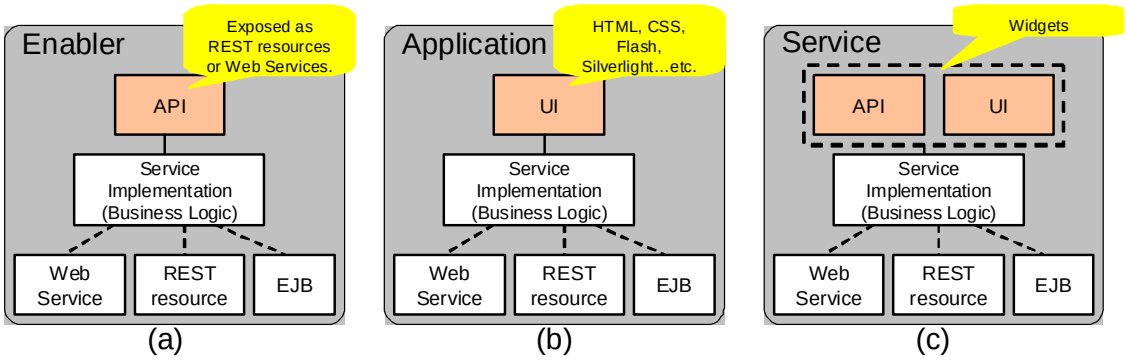


Figure 34: Service components.

Unlike enablers, applications (Figure 34.b) are designed to be used exclusively by users. Therefore, applications are not described in a machine readable language; instead the focus is on its interaction with the user. As a consequence, several technologies enable today the creation of advanced UI such as HTML/CSS, Adobe Flash, or Microsoft Silverlight. In addition to this presentation layer, applications also include the business logic which is implemented using any programming language such as Java and PHP.

Finally, the conceptual distinction of a Service (Figure 34.c) is its characteristic of being designed to be used either by another (third-party) software entity, or directly by users. Therefore, it is important to focus its technical realisation on both UI aspects and reusability capabilities. Consequently, we technically define a Service as a software entity realizing a business operation; it is associated to a UI (Widget), which enables the user to interact with the business operation and vice versa (see Figure 34.c); the Widget part should be described and should include semantic annotations in order to enable machines to reuse the associated software service. By supporting both users and machines, the Widget paradigm in this thesis fills the existing gap between the concepts that are understandable only by the user (UI, service), and those that are understandable only by machines (XML, HTTP…etc).

2 Widget Oriented Architecture (WOA)

Our goal in this thesis is to propose solutions to current SOA limitations. We define thus the WOA (Widget-Orient Architecture); a new paradigm which is more user-centric than SOA. Similarly to SOA, WOA is a new computing paradigm that utilizes Widgets as basic elements to support the development of rapid, low-cost and easy composition of distributed applications even in heterogeneous environments. As we previously defined, a Widget is the UI part of a service that includes semantic annotations to enable machines to process it. Our goal is to enable the reuse of the service by both humans and machines.

As illustrated in Figure 35, WOA includes five roles: a Widget provider, a registry, a Widget client, a Widget developer, and a user. The Widget provider publishes Widgets to a common registry by providing their description, annotated semantically using the platform semantic dictionary. The Widget provider is not necessary the developer of the Widget, neither the developer of the corresponding business logic. The user uses the Widget client to discover, load, use, and compose Widgets. All interactions between the user and the service go through the Widget client. The Widget client interacts with the registry and the Widget provider. It invokes Widgets through HTTP request; the response is a UI (XHTML code) dynamically generated by the service implementation, and semantically annotated.

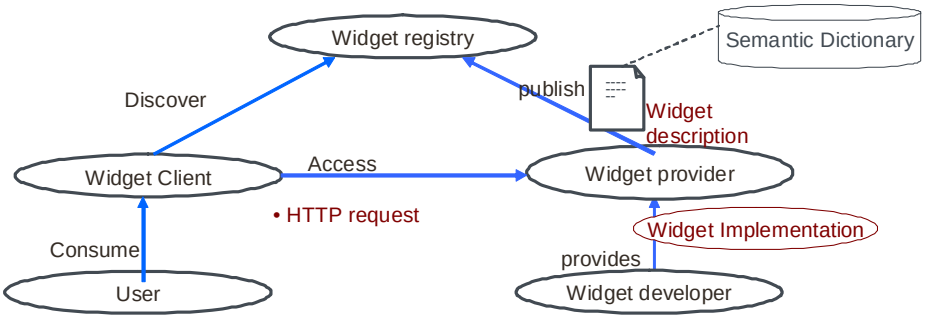


Figure 35: Basic Widget-Oriented Architecture.

Similarly to SOA, WOA relies on a set of principles which are categorized into Widget registry principles, Widget client principles, and Widget provider/developer principles. After reviewing in the following subsections these principles, we summarize how the different actors involved in the WOA model interact with each other.

2.1 Widget Registry

Such as SOA service registry, the Widget registry must provide an interface for publishing Widgets and another for discovering Widgets. However, with the increasing number of services, it is likely to have several Widgets fulfilling the same functional need. Therefore, it is important to provide a mechanism for selecting services among functionally equivalent ones. This selection mechanism must be user centric; in other words, it must be based on selection rules defined by the user himself.

2.2 Widget Client principles

The Widget client is a software application through which the user consumes Widgets. Followings are a set of principles related to this role.

f. *Service Environment as a Composition Framework*

The Widget client must play a role of a user service environment and a composition framework at the same time. In other words, in WOA paradigm, the composition framework should be the same environment as

the daily service environment of the user. The user should not have two separate environments, one for composing services and one for using the composition.

g. Widget Client Personalization

It is important to enable the user to personalize his service environment according to his own needs and preferences. He should be able for instance to add and remove Widgets from his environment, and to compose the Widgets with each others.

h. Widget Discovery

The Widget client should provide users with a UI that enables them to interact with the discovery interface of the Widget registry. Thus, in addition to the functional based discovery, the Widget client should enable the user to specify the rules to apply to select services among functionally equivalent ones.

i. Reusability and Composition of Widgets at the UI level

Such as in SOA, reusability and composition are main principles in the WOA. However, WOA distinguish itself by being closer to the users. The reusability and composition must be performed at the UI level (as illustrated in Figure 36). In other words, the intelligence that enables the composition of Widgets between them must reside at the Widget client level. This enables the design of user-centric mechanisms as we will show later on in this thesis.

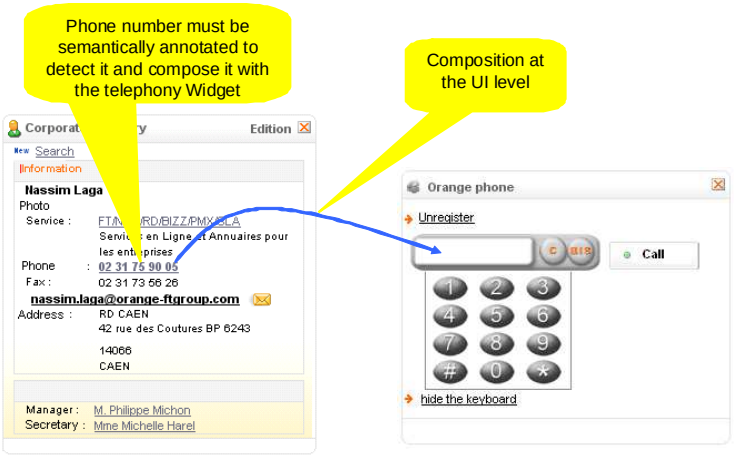


Figure 36: Reusability and composition at the UI level.

In addition to the reusability and composition of Widgets within the Widget client level, it is recommended to consider two additional issues (to enhance even more the user centricity of the WOA paradigm): reusability cross Widget clients (see Figure 37), and reusability based on unstructured data (see Figure 38). First, with the proliferation of devices, the users would likely want to combine services loaded on different devices. The reusability and composition should therefore be possible even between two

Widgets loaded on two different Widget clients (optionally on different devices of the same user as illustrated in Figure 37).

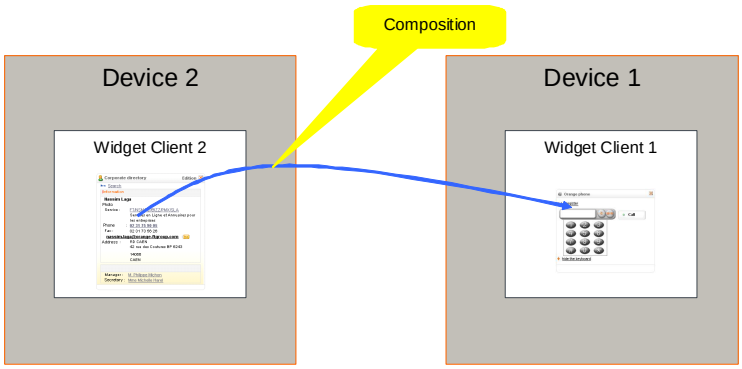


Figure 37: Reusability across different Widget clients.

Second, with the proliferation of communication services, much data (e.g. phone numbers, email addresses, postal addresses…etc) are generated and exchanged between users; data that could be used as input parameters in the invocation of other Widgets. As in the WOA we are acting at the UI level, it is pertinent to provide tools for capturing these data and composing them with other Widgets. As illustrated in Figure 38, this would enable for example the capturing of postal addresses within an email and locating them using a Map Widget.

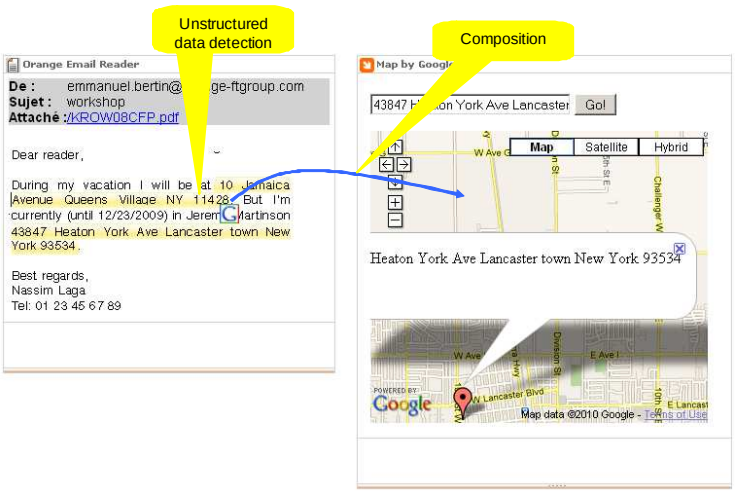


Figure 38: Unstructured data composition.

j. *Stateless and Statefull*

Unlike SOA which recommends stateless services, Widgets in WOA can be statefull. Indeed, as Widgets embed a UI that interacts with the backend logic, they can manage different states at the UI level without affecting the performance of the backend service implementation.

It should ne noted that the capabilities of a Widget may differ from a state to another. For example, a telephony Widget, in its initial state, can make and receive calls; but, the same Widget does not have that capabilities when a call has been established. Therefore, it is important to enable Widgets to subscribe and unsubscribe capabilities at runtime, during their lifecycle within the Widget client.

2.3 Widget provider/developer principles

a. *Widget-based Development of Services*

In WOA, developers expose their applications as a set of Widgets. A Widget provides access to one, and only one, service implementation. Ideally, a service implementation embeds one functionality (a software entity that provides an added value for users). But in some cases, it is necessary to include several functionalities within the same service implementation to enhance the user experience. For instance, consider a telephony service. It embeds the functionality that enables the users to make calls, and the functionality that enables them to receive calls. From the user point of view, it is not necessary to have two separate UIs. Thus, for the sake of the user convenience, two or more functionalities could be included within the same UI. This is analogous to Web services which embed several operations within the same Web service. Figure 39 summarizes how applications are exposed as a set of Widgets.

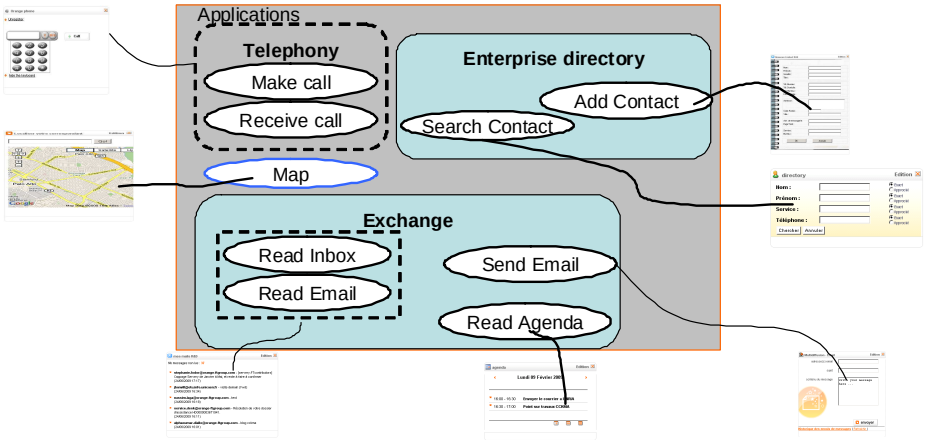


Figure 39: Exposing applications as a set of Widgets.

b. *Contracting*

Such as SOA, WOA requires from Widget providers to define their Widgets in term of the functionalities they provide, and their non-functional parameters (including the provider, the version, the SLA, and the Qos…etc).

c. *Semantic Typing of the UI*

The Widget UI enables the user to interact with the underlying business logic, which includes entering the required inputs, invoking the needed functionality, and displaying the results for the user. In WOA, it is

required from Widget developers to semantically annotate the results at the UI level. This is important for reusability and composition (e.g. to use that result in the invocation of functionalities of other Widgets); this is detailed in *Chapter II.2 A Design of a Widget-Oriented Architecture (WOA)* and *Chapter II.3 Widget-Oriented Architecture (WOA) in SOA application fields*. Figure 36 illustrates this principle.

d. Widget Autonomy and Loose Coupling

Such as services in the SOA, Widgets in the WOA must be autonomous and do not depend on an external system. Each Widget must have its own lifecycle, and depend only on the business logic it implements. The autonomous aspect of services and Widgets promotes reusability. However, as Widgets are designed also for users, the Widget clients and the Widget providers must rely on the same Widget API.

2.4 Interactions

There are five interactions between the different actors involved in the WOA model. Table 8 summarizes them.

Table 8. Interactions of WOA model actors (Figure 35)

Interaction Name	Involved actors	Description
Provide	Widget developer – Widget provider	The developer is in charge of developing the Widget source code and providing it to the Widget provider, to host it. It is important for the Widget provider to know the functionalities of the Widget, in order to describe it and publish it to the registry.
Publish	Widget provider – Widget registry	The publication phase could be performed by providing a file compliant to the syntax and the semantic defined by the Widget registry, or automatically by filling a form, in which case the file will be created automatically. In both cases, it should be provided the index URL of the Widget, the URLs of the functionalities of the Widget, their inputs, and their outputs.
Discover	Widget client – Widget registry	The discovery interaction aims to retrieve the Widgets present in the registry according to a given criteria. This implies an agreement on the format and the interface that must be used between the two entities. From the technical

		perspective, this could be performed using for instance a REST API. The response of the Widget registry is a list of Widgets. Each entry of this list must contain the index URL to invoke the Widget, and the list of functionalities and their URLs.
Access	Widget client – Widget provider	The Widget client loads a Widget by invoking the corresponding index URL (retrieved at the discovery step). The response to this invocation is a UI (e.g. HTML/CSS). When the Widget client needs to invoke a functionality (when composing Widgets for example), it must invoke the corresponding URL with the correct parameters provided by the Widget provider when publishing the Widget.
Consume	User – Widget Client	The user interacts with the Widget client through the Widget client UI. This UI should provide to the user the capability of discovering, using, and composing Widgets.

Chapter II.2 A Design of a Widget-Oriented Architecture (WOA)

In this Chapter, we design a Widget aggregator framework which is compliant with the principles of the WOA paradigm we defined. We first detail how we model a Widget. Second, we summarize the high level view of the Widget aggregator we propose. Third, we detail the most important innovative mechanisms we define.

1 Widget

As we previously specified, a Widget is basically a UI that provides access to a service implementation; where a service implementation may provide several functionalities. The description of these functionalities is a must in the WOA paradigm. In addition to that, their outputs, which are displayed in the UI, must be semantically annotated. Figure 40 illustrates the Widget model we propose in our architecture. Some of the defined elements already exist in current Widget paradigm (white part), and some others are new (grey part) (see Figure 28 to compare).

Such as traditional Widget model, in our architecture each Widget has an implementation and a description file (contract). The implementation part implements different modes (at least a view mode). However, In addition of modelling and describing the different modes of a Widget, we also consider the functional and the non-functional view of these Widgets. Each Widget may provide one or several functionalities. These functionalities are described within the Widget description file. Each functionality description contains an abstract description part and a concrete description part. The abstract description part describes the goal of the functionality, the inputs it requires, and the outputs it generates. The functionality goal, the inputs, and the outputs are described using a semantic dictionary provided by the Widget aggregator (Widget client).

The concrete part of the description file refers to the actual implementation of the Widget. For each functionality declared, the provider must specify the URL that provides such functionality. For each input and output of a functionality declared, the provider must specify the corresponding tag used within the actual implementation of the Widget. This enables decoupling Widget developers and the semantic dictionary used within the Widget aggregator; the developers are not obliged to use the semantic dictionary

of a specific aggregator; instead, the mapping between their tags and the Widget aggregator's ones is performed by the Widget provider in the description file of the Widget.

The non-functional view of a Widget is described through a list of parameters and their values (e.g. (provider, Orange)).

Another requirement of WOA is the annotation of the Widget outputs at the UI level. This is also modelled in Figure 40, in which outputs contained within a UI include a data type (the annotation) and the data value.

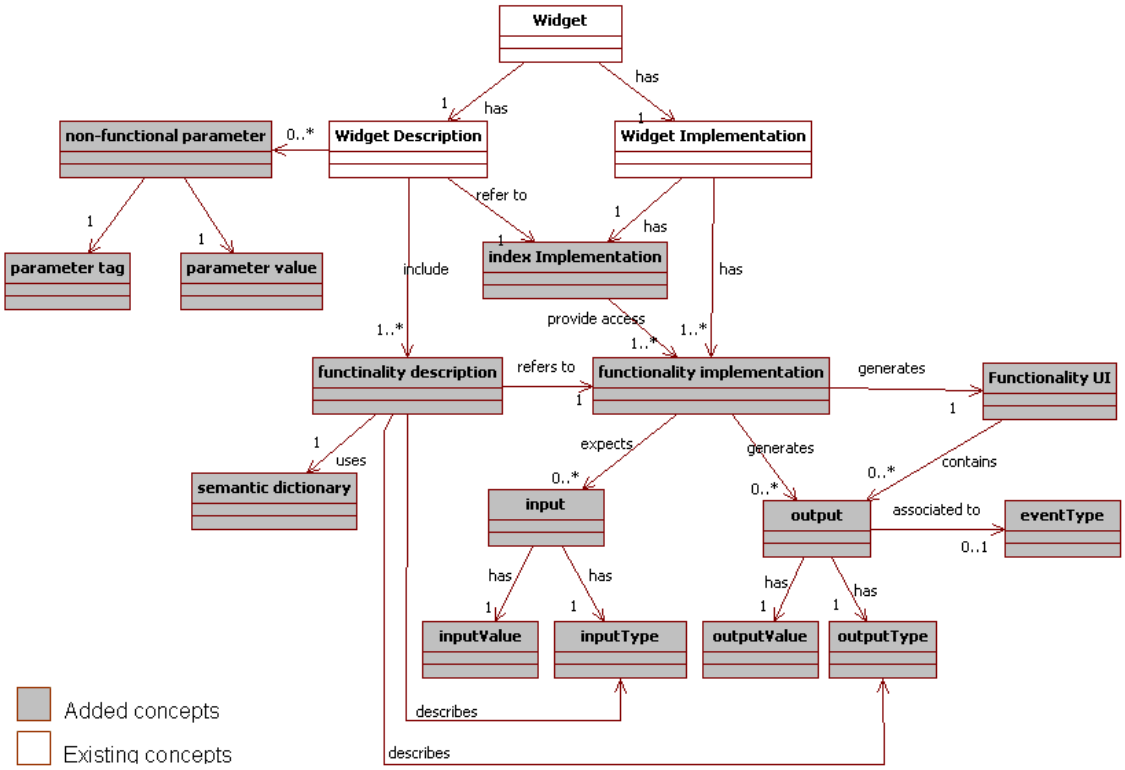


Figure 40: Widget design.

2 Widget Aggregator

The Widget aggregator plays the role of a Widget client in WOA paradigm. It is mediating between users and Widget providers. In WOA paradigm, the Widget client must embed a set of features, namely: a user-centric Widget discovery capability, reuse and composition of the Widgets at the UI level, supporting Widgets reusability and composition across different Widgets clients, and finally the detection and composition of unstructured data. In this section we provide a high level view of the architecture of the Widget aggregator. Then, in the next section, we detail the Widget Combination component, which is our main contribution as it enables the Widget reuse and composition at the UI level; it supports Widgets

reusability and composition across different Widgets clients; and finally, it supports unstructured data based composition.

The Widget aggregator we propose is characterized by 6 main components: the GUI component, the Widget Combination component, the Persistency Manager component, the Interpreter component, the Widget Registry component, and the Semantic Dictionary component. As illustrated in Figure 41, some of these components run at the front-end, while others run at the backend.

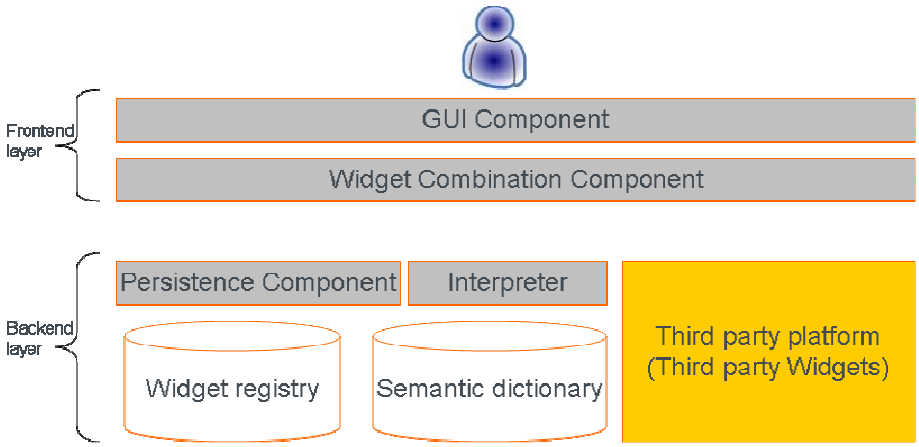


Figure 41: Service aggregation high level architecture.

The GUI is the visual representation of the framework. It provides access to the Widget registry and its discovery capability; it enables the user to load a Widget and use its functionalities; and it enables him to access to the Widget combination capability.

The Widget Combination component characterizes our main technical contribution. It provides the reuse capability (a WOA principle) to users and developers. This component is detailed in the following section.

The Persistency component is in charge of interacting with the Widget registry and the semantic dictionary when needed. The Widget registry stores Widget description files and should provide a discovery interface, which is used by the frontend components (GUI and Widget combination component). These components (Persistency component, Widget registry, and the semantic dictionary) run at the backend as they deal with persistency.

The Interpreter component provides runtime Widget selection mechanism. This component provides a user centric Widget selection mechanism used to first decouple composite services from the basic services they use; and second, it enables a dynamic adaptation of composite services according to new contexts. This component is detailed in section *3.4 Abstract Service Based Reuse Extension*.

3 WOA Key Functionalities (Widget Combination Component)

As illustrated in Figure 42, the Widget Combination Component provides three mechanisms that enable the reuse of Widgets functionalities:

- API based (developer oriented) reuse of Widgets.
- automatic and semantic based reuse of Widgets (user oriented),
- and process based reuse of Widgets (user oriented),

In addition to these three innovative mechanisms, the WOA we propose is also extended with three main concepts in order to enhance it and resolve SOA limitations that are still not resolved with the basic WOA. These three mechanisms are:

- abstract service based reuse of Widgets,
- unstructured data based reuse of Widgets,
- and cross-device reuse of Widgets.

In the following subsections, we detail each mechanism and each extension.

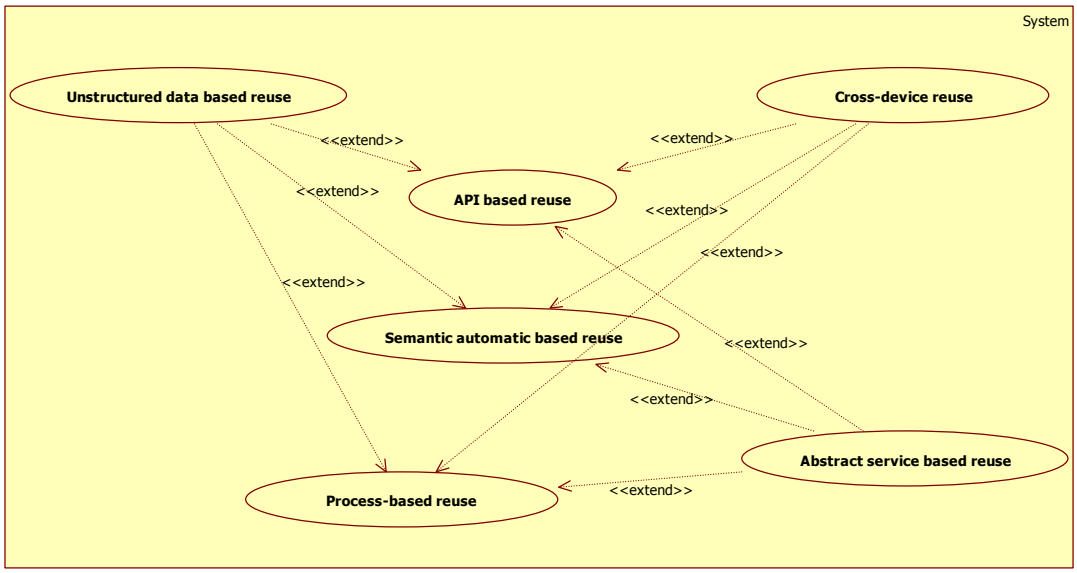


Figure 42: Use case view of the Widget Combination component.

3.1 API-based Reuse of Widgets

a. Mechanism Goal

One approach for performing Widget reusability is to rely on the *API-based reuse of Widgets* mechanism. This mechanism is conceived for Widget developers in order to enable them to reuse capabilities of other Widgets loaded to the user environment. The specificity of this mechanism, compared to traditional ones

(mainly SOA APIs), is characterized by the user centricity of the reuse approach. The mechanism provides an API that enables developers to discover and reuse capabilities of Widgets that are loaded to the user environment at runtime; whereas in SOA, developers discover and use capabilities of services present in the registry, which do not necessary have any relationship with the user. Figure 43 shows the difference between the two approaches.

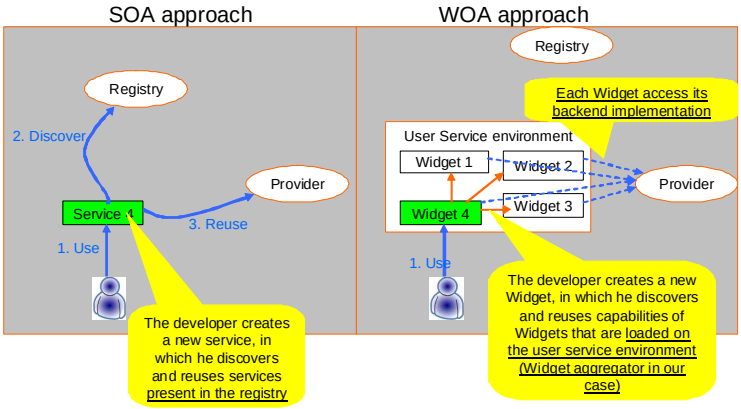


Figure 43: SOA approach vs WOA approach in API-based reuse.

b. Mechanism Design

This mechanism is characterized by the API component illustrated in Figure 44. This component has the advantage of being related to the Widget aggregator instance; which provides some user context information. It can be summarized in four main functions: *GetWidgetList*, *Subscribe*, *Unsubscribe*, and *Publish*.

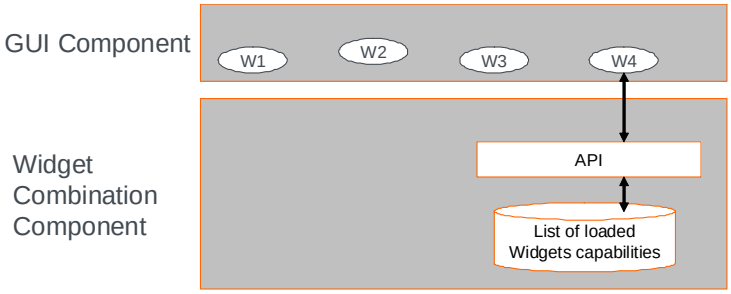


Figure 44: API-based reuse involved components

- The *GetWidgetList* function provides the list of Widgets that are loaded within the Widget aggregator instance, which is directly related to a user. The discovering step could be done either according to the functionality that a developer needs, or according to a given parameter type, which can be used as input parameter of the discovered Widgets. The functionality and the parameter type are described using a tag defined in the semantic dictionary.

- The *Subscribe* function enables the Widgets to declare their functionalities; this can be performed automatically by reading the Widget description file, or explicitly by the developer of the Widget by using the API.
- The *Unsubscribe* function enables the developer to declare the unavailability of a capability previously declared. The *Subscribe* and *Unsubscribe* functions together enable the modification of Widget capabilities at the runtime.
- Finally, the *Publish* function enables a Widget developer to use capabilities of other Widgets loaded within the same user environment (Widget aggregator instance). The generated outputs of the source Widget are thus used as input parameters of the destination Widget(s); the outputs are explicitly specified as an argument of the *Publish* function.

As illustrated in Figure 44, the *GetWidgetList*, *Subscribe*, and *Unsubscribe* functions enable developers to read and write on the list of capabilities of the Widgets loaded on the Widget aggregator.

3.2 Semantic and Automatic Based Reuse of Widgets

a. Mechanism Goal

The semantic and automatic based reuse of Widgets is another approach to perform Widget reuse; one principle of the WOA paradigm. It is designed for users and characterized by automatically discovering at runtime the capabilities of the Widgets loaded on the user environment, and composing them if a semantic matching is detected. The discovery and the reuse must be performed by the Widget aggregator, and not by the developer of a specific Widget as it is the case in the API-based reuse of Widgets. Figure 45 illustrates this mechanism.

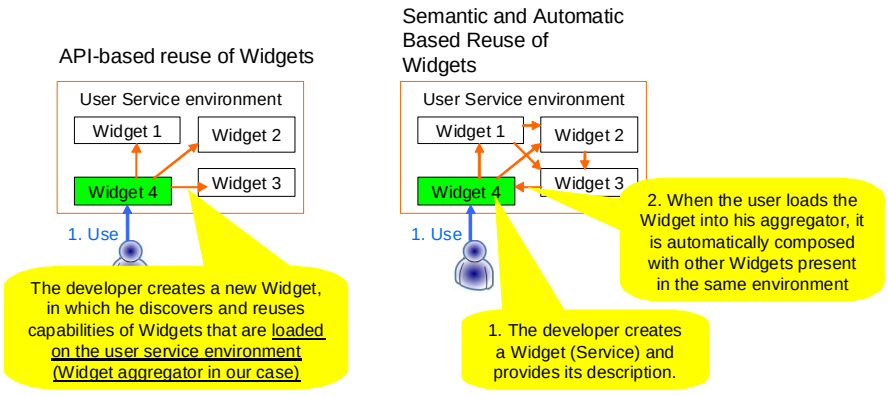


Figure 45: Semantic and Automatic Based Reuse of Widgets Summary.

b. Mechanism Design

To automatically combine Widgets, this mechanism relies on the Widgets descriptions (contracts). As Widgets are described in term of the functionalities they provide, the type of inputs expected by each functionality, and the type of outputs they would generate, the Widget combination component, incorporated at the Widget aggregator level, can easily detect the semantic compatibilities between Widgets instances that are loaded on the user environment. Then, for each semantic compatibility detected between two Widgets, the Widget combination component creates a link between them; enabling by consequence the user to easily combine the two Widgets. This is performed through a sub-component named "Communication Manager" illustrated in Figure 46.

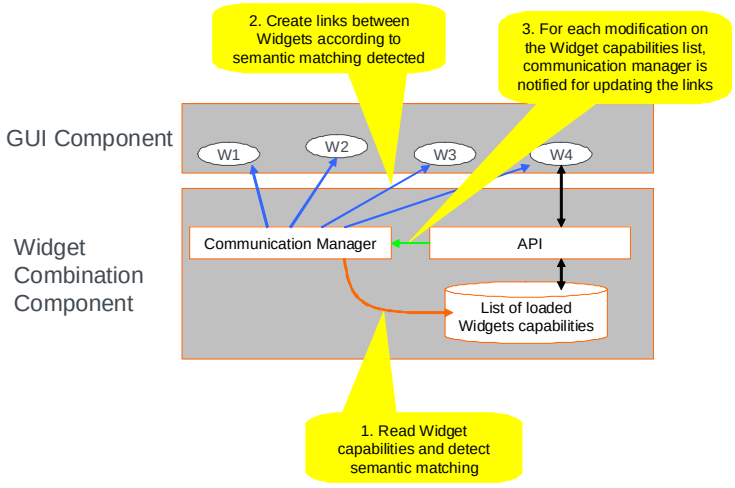


Figure 46: Communication Manager component.

The semantic concepts are described using *microformats* paradigm. The semantic matching between an output of a Widget and an input of another follows one of the three patterns illustrated in Table 9; we associate for each one a set of actions:

Table 9. Semantic matching patterns.

Pattern name	Pattern description	Actions
Exact Matching	The output type (of the source Widget) is exactly the same as the input type (of the destination Widget)	The only action performed when detecting such semantic matching is the creation of the link between the two Widgets; when the link is executed, the output of the source Widget is transmitted as input to the destination Widget without any modification.
Inclusion	The output type (of the	A link between the source and the destination Widget is

	source Widget) is a sub-element of the input type (of the destination Widget)	created. At the runtime, the output of a source Widget is transformed to the format of the input of the destination Widget (the other elements that construct the input of the destination Widget are empty). The result is transmitted to the destination Widget as an input parameter when the link is executed.
Reverse inclusion	The input type (of the destination Widget) is a sub-element of the output type (of the source Widget)	A link between the source and the destination Widget is created. At the runtime, the input type is extracted from the outputs generated by the source Widget. It is transmitted to the destination Widget as an input parameter when the link is executed.

The usage of microformats instead of ontologies is motivated by the fact that the combination process is performed at the presentation layer. The presentation layer enables the framework to harness the user intelligence to make up the lack of microformats semantic expressiveness. First, the probability of detecting wrong semantic matching is significantly reduced as the user would likely load Widgets of the same business domain; and second, the user can check whether two detected semantic matching are really compatible at the business level or not.

Links between Widgets are defined with a sextuplet *L (Source-Widget, Output-Type, Destination-Widget, Destination-Functionality, Input-Type, Link-Type)*. There are two types of links between Widgets: automatic links and semi-automatic links. Automatic links are executed without any initiative from the user. Each time the data and/or event that should be transmitted from the source Widget to the destination Widget are detected, the destination Widget is automatically launched without any direct initiative from the user. Semi-automatic links are however first displayed within the UI of the source Widget using HTML elements (typically an icon). Then, when the user clicks on that HTML element, the corresponding data are transmitted from the source Widget to the destination Widget.

3.3 Process-based Reuse of Widgets

a. Mechanism Goal

The Communication Manager component, introduced in the previous section, creates links between Widgets based on semantic matching of microformats tags. One limitation of such approach is that it may lead to an environment where some created links are intrusive and undesired by the users. This is especially due to the lack of semantic expressiveness of microformats. However, as the reuse and composition

mechanisms are implemented at the presentation layer, it is possible to easily interact with the user and harness his intelligence. Therefore, we propose in this section the *process based reuse* mechanism. First, it aims to provide a controlled approach when reusing Widgets, by relying on a flowchart definition. Second, it enables the user to modify that flowchart in order to delete undesired links for example. Figure 47 shows the difference between the process based reuse of Widgets mechanism, and the semantic and automatic reuse of Widgets mechanism.

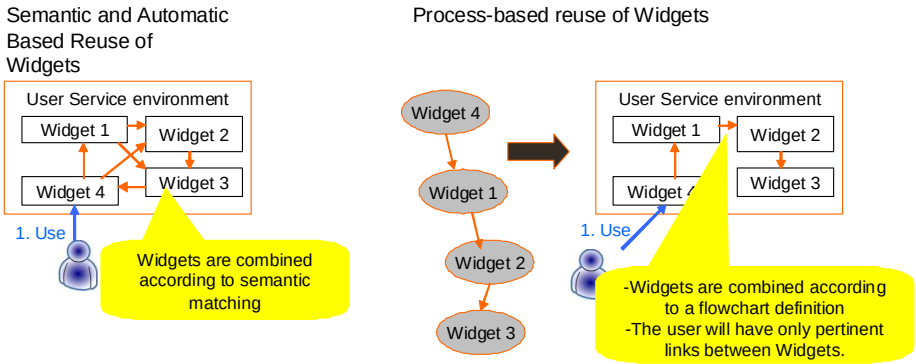


Figure 47: Process-based reuse of Widgets goal.

b. Mechanism Design

In order to provide a controlled approach when reusing Widgets, we introduce the "Process Manager" component which relies on a flowchart (process) definition to enable the reuse of Widgets. Figure 48 shows the Process Manager component within the architecture of the Widget aggregator. Thus, users, or service aggregator provider, may define a process which controls the Widget combination. In other words, only Widgets combined within the process definition are actually combined at the execution time (within the GUI level).

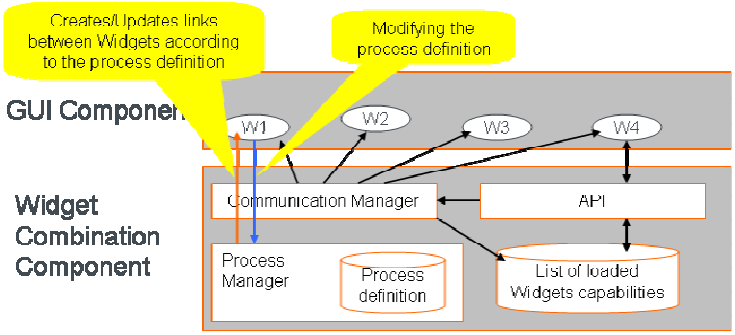


Figure 48: Process Manager Component.

We define a process as a graph $G <N, L>$, where nodes N represent the list of Widgets that are loaded, or must be loaded, on the user environment; and links L represents the links between the different Widgets. There are two types of links: automatic, and semi-automatic. In an automatic link, the communication between the two corresponding Widgets occurs each time the corresponding data are

available at the source Widget. In semi-automatic links, the communication occurs only at the user initiative. Each link in *L* is defined by sextuplet *L (Source-Widget, Output-Type, Destination-Widget, Destination-Functionality, Input-Type, Link-Type)*, where *Source-Widget* – an element on *N* – is the source Widget of the link; *Output-Type* is the type of a *Source-Widget* output; *Destination-Widget* – an element of *N* – is the destination Widget; *Destination-Functionality* is the functionality of *Destination-Widget* which should be invoked; *Input-Type* is the input parameter of *Destination-Functionality*; and finally, *Link-Type* is the type of the link (automatic, or semi-automatic).

At the execution time, the Process Manager component relies on a process definition to define which Widget is linked to another. In addition, it provides an interface that enables the modification of the process definition at runtime. This interface should be used by the GUI by providing intuitive GUI elements that enable the user to modify the process definition.

3.4 Abstract Service Based Reuse Extension

a. Extension Goal

The goal of this extension is to define a composition of functionalities, which are associated at the runtime to Widgets. However, several Widgets may fulfil the same functional need. Therefore, the runtime selection of the best Widget for a given functionality is a challenge; especially, when the selection criteria are different from a user to another, and from a functionality to another. The abstract service based reuse is a mechanism that aims to respond to that challenge. It extends the previously defined reuse mechanisms by providing a runtime Widget selection mechanism. The specificity of our approach is the user centricity of the selection process. Indeed, the selection process relies on rules that are defined by the user.

This mechanism promotes loose coupling between services (Widgets). Indeed, as compositions refer to functionalities, they remain completely independent from the Widgets that are really invoked.

b. Extension design

The solution we propose to provide such extension relies on two main components: the abstract Widget, and a dynamic selection mechanism. Figure 49 shows these two components within the global architecture.

The Abstract Widget is a Widget defined by the Widget aggregator provider (Widget client), which is associated to a functionality, and a list of selection rules that could be applied on that functionality. The selection rules define the criteria to apply in the process of selecting a Widget for performing the associated functionality. The selection process is performed by the Interpreter component illustrated in Figure 49. The GUI of the abstract Widget should:

- enable the user to enter the values of the inputs required by the corresponding functionality,

- enable the user to choose which selection rules to apply in the selection process,
- invoke the Interpreter component to select the best Widget,
- display the selection results of the Interpreter,
- and invoke a selected Widget and display the results.

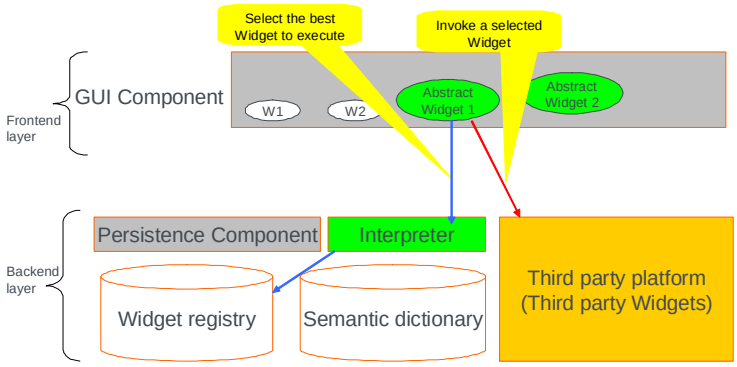


Figure 49: Components involved in the abstract service based reuse extension.

Figure 50 shows the different concepts involved in the design of the abstract service reuse extension. As we previously mentioned, an abstract Widget associates a functionality (e.g. *Get_News*) to a set of selection rules (e.g. select the Widget according to my preferred language). A functionality is defined with its goal concept (e.g. *Get_News*) – a semantic concept is a tag defined within the semantic dictionary, and the type of inputs it expects and outputs it generates. Selection rules represent the logic followed by the Interpreter component in the selection of the best Widget for satisfying the goal concept. We define two types of selection rules: constraint rules and objective rules. A constraint rule is a condition that a selected Widget must satisfy (e.g. the price does not exceed 5 Euros). An objective rule is an optimization of a given function (e.g. minimizing the price, or minimizing a linear objective function). Each selection rule can refer to two types of parameters: static parameters and dynamic parameters. Static parameters refer to any parameter which does not change frequently, and whose value is stored in the Widget aggregator database (e.g. price). Static parameters are accessible through a specific component named knowledge base. Dynamic parameters refer to any parameter whose value is known only at runtime. This includes the user inputs and other parameters such as the location and the presence of a user, which are generated by other services (e.g. localization enabler). The specificity of this model is the capability of specifying rules that refer to heterogeneous parameters. The only condition required is the ability of generating the value of that parameter by a service present in the registry (e.g. location, and presence), or the presence of the parameter value in the Widget aggregator database (e.g. price of a Widget).

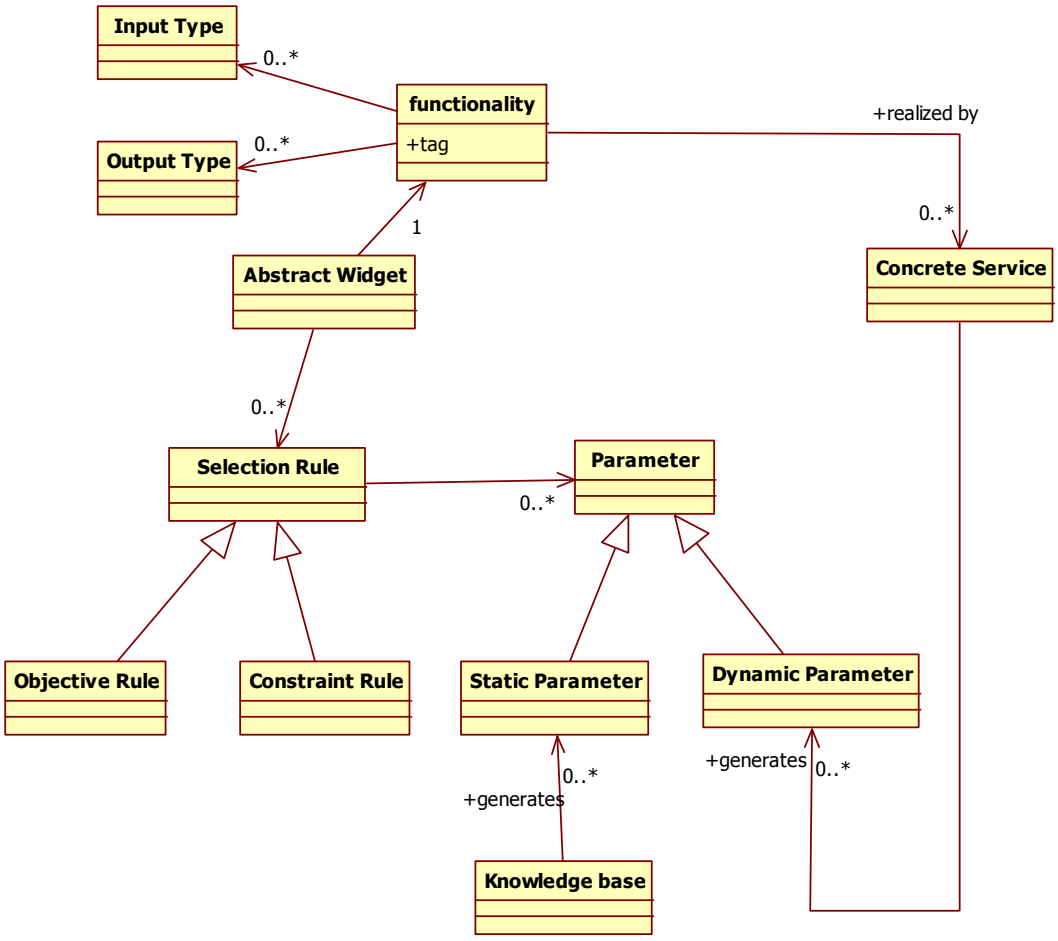


Figure 50: Abstract Widget related concepts.

In order to select the best available concrete Widget that responds to an abstract Widget (realize the functionality, satisfy the constraint rules, and optimize the objective rule), we introduce the Interpreter component (see Figure 49). It receives as input parameter a functionality, the required functionality inputs, and a set of selection rules that are activated by the user. It generates a list of concrete Widgets. As we illustrate in Figure 51, the first action carried out by this component is the discovery of all available concrete Widgets that performs the functionality of the abstract Widget. Thereafter, the discovered Widgets are filtered according to a set of constraint rules. Each constraint rule may refer to a static or a dynamic parameter. The dynamic parameters are computed at the runtime by invoking the corresponding services (e.g. invocation of the localization service to get the user location parameter). Once all constraint rules are applied and a set of concrete Widgets are selected, the Interpreter evaluates the objective rule if present. Such as constraint rules, the objective rule may refer to static and dynamic parameters; the dynamic

parameters are computed at the runtime. At the end, a set of concrete Widgets are selected; concrete Widgets that satisfy the constraint rules and optimize the objective rule.

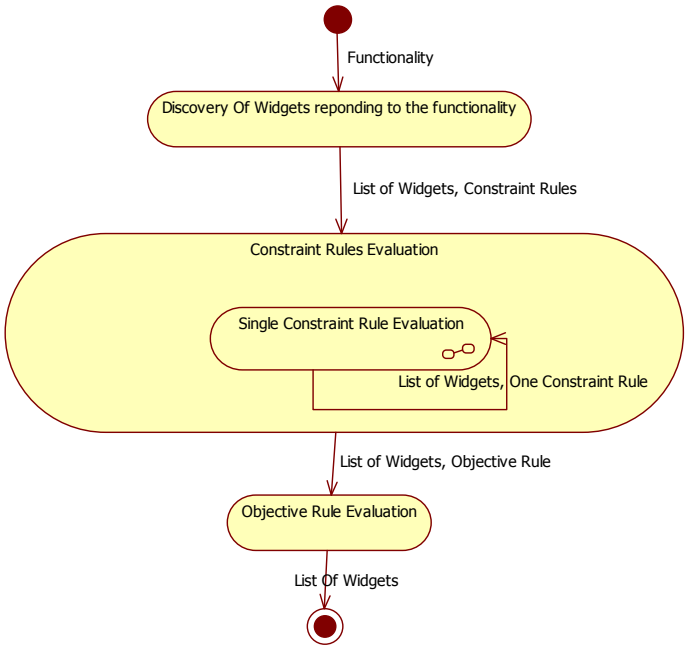


Figure 51: Service selection algorithm.

The Interpreter component is not specific to the WOA. It can be used in the traditional SOA to perform runtime service selection.

The abstract Widget enables the users not only to enter the inputs expected by the corresponding functionality and display its outputs, but also to choose the selection rules they want to apply in the selection process. We define thus a user centric service selection mechanism.

The format of the Abstract Widget is exactly the same as ordinary Widgets, previously defined. Thus, in the Edition mode, the Widget provides the UI that enables the user to choose (check and uncheck) the selection rules they want to apply. The View mode displays the UI that enables the user to enter the inputs expected by the corresponding functionality, it invokes the selection process (Interpreter component), it displays the result of the selection process (Interpreter outputs), and it invokes a selected Widget and displays the result.

In addition of providing a user centric selection mechanism, the Abstract Service based reuse extends the previous reuse mechanisms. First, it provides a goal based reuse; and thus decouples the Widget providers from the Widget integrators (those that combine Widgets with each other, including the

user). Second, it provides a dynamic adaptation according to users criteria. Figure 52 summarizes this extension.

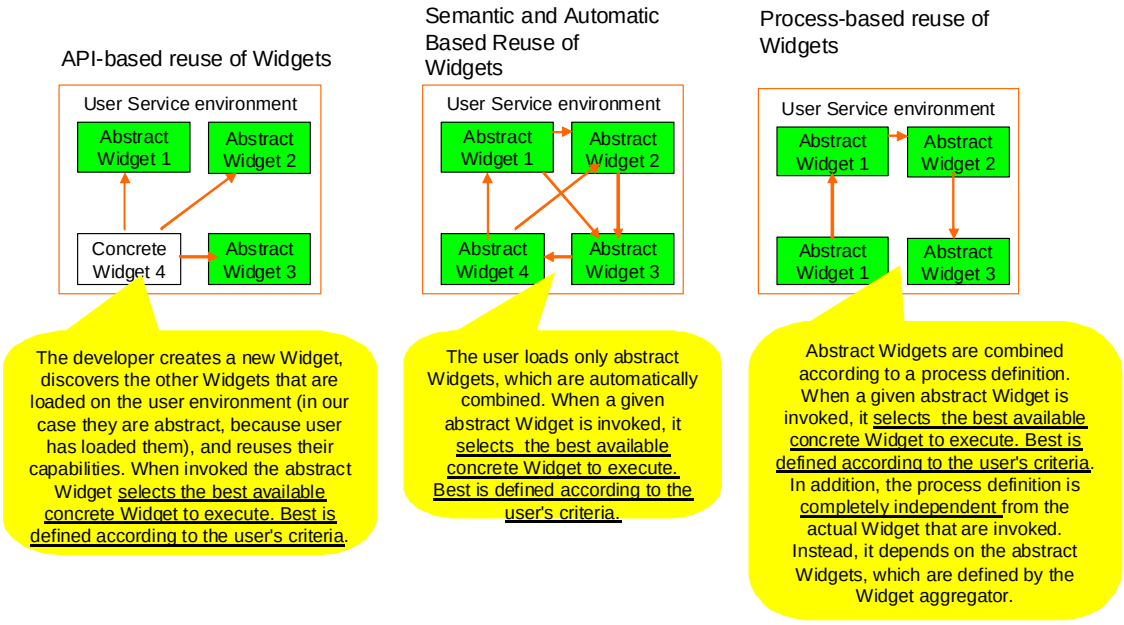


Figure 52: Illustration of the abstract service based reuse extension.

3.5 Unstructured Data Based Reuse Extension

a. Extension Goal

The most important reuse pattern in SOA is characterized by capturing outputs of a source service and sending them as input parameters to a destination service. Such mapping currently requires from service developers to declare the inputs expected by the functionalities they provide, and the outputs generated by these functionalities. This declaration might be performed at the publication time, or at the runtime as we previously proposed using a *publish* function. In both cases, it is required from the developers to know all data that can be generated by the services they provide. In addition, developers of two composed services must rely on a common semantic in their service descriptions.

However, services in current Web platform are likely to generate data that are not expectable by their developers, and thus impossible to include in the service description. This is especially true when considering Web 2.0 paradigm from one hand, and the growing number of communication services from another hand. Indeed, Web 2.0 promotes user generated content; content which includes text-based data (such as Wikipedia) and multimedia-based data (such as Flickr and YouTube). That data might contain information that would be useful to reuse in other services, but hardly expectable and manageable by service developers. In addition, people are more and more connected together. Thus, a great amount of data

is generated and exchanged between them; data that also might contain information (e.g. phone numbers, addresses…etc.) that would be useful to reuse in other services, but not expected by service developers. Consequently, much data are unavoidably discarded by current SOA.

The extension we propose in this section aims to tackle this limitation. Our goal is to enable service integrators (users and developers) to first capture the useful and unstructured data; and second, to reuse them in other services.

b. Extension design

The mechanism we propose in this section is an extension to the previously detailed reuse mechanisms, namely API-based reuse, semantic and automatic based reuse, and the process-based reuse. This extension is characterized by:

- The integration of a new entity to the Widget-oriented architecture. This entity is a repository that contains data extraction modules. These modules, when invoked, are in charge of extracting unstructured data from a specified Widget. These modules do not provide any added value for users, but they enable service integrators to extract, and make use of, unstructured data.
- The definition of a new service reuse pattern that facilitates the reuse of unstructured data generated by a Widget within another one.

As illustrated in Figure 53, we introduce new roles to the WOA: the provider of the data extraction modules, and the registry of the data extraction modules. The data extraction modules provider is in charge of creating the unstructured data extraction modules, and publishing them into the data extraction modules registry. The unstructured data extraction modules must be defined in conformance of an API defined by the Widget aggregator (in our case, they must be defined using JS language, and implement "*extract_Data*" function). The publication process is performed by providing a description file. The description file contains mainly a tag representing the type of data that can be extracted by the module; the tag refers to a concept within the semantic dictionary. More precisely, it refers to a data type. It is recommended to have one data extraction module for each data type. This enables the platform to deduce the data extraction module to invoke according to the type of data to be extracted.

In practice, the provider and the registry of the data extraction modules can be the same entity as the Widget Client provider.

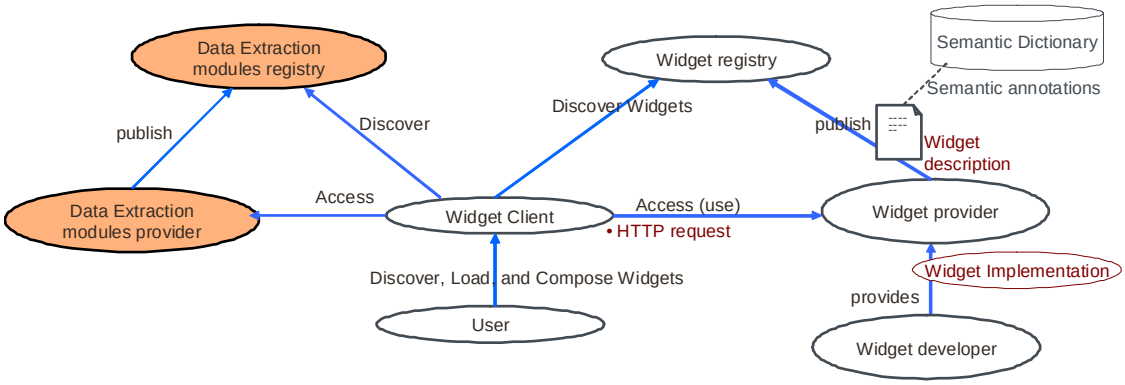


Figure 53: Unstructured data based composition – architectural model.

The first basic approach to harness the unstructured data based reuse extension is to manually use the data extraction modules and to manually invoke the Widgets. Thus, service integrators (mainly developers) first discover the source and the destination Widget. Second, if the inputs of the destination Widget are different from legacy outputs of the source Widget, developers invoke a data extraction module that enables them to extract the input needed by the destination Widget from the output generated by the source Widget. Third, developers invoke the destination Widget using the extracted data as input parameters. The invocation process might be performed either through the Widget Combination API (previously detailed), or by connecting directly to the Widget provider. This process can be automated with a service composition language. This is detailed in the next Chapter.

The unstructured data based reuse mechanism extends the API based reuse by providing developers with a function that enables them to easily associate a data extraction module to their Widgets. When the corresponding data is detected, a call back function (defined by the Widget developer) is invoked. In that function, the developer can discover and invoke Widgets that are loaded in the user environment and that can handle the extracted data. Figure 54.a shows this mechanism.

The unstructured data based reuse mechanism extends the semantic and automatic based reuse by first detecting the inputs required by the functionalities of the Widgets loaded in the same user environment; second, associating the corresponding data extraction modules to the different Widgets; and third, creating links when that data are detected and extracted. This is illustrated in Figure 54.b.

The unstructured data based reuse mechanism extends the process based reuse of Widget by providing the capability of defining processes based on unstructured data. In other words, it enables the definition of links where the output data should first be detected within the Widget, second extracted, and then transmitted to the destination Widget as input parameters. This is illustrated in Figure 54.c, and detailed in the next Chapter.

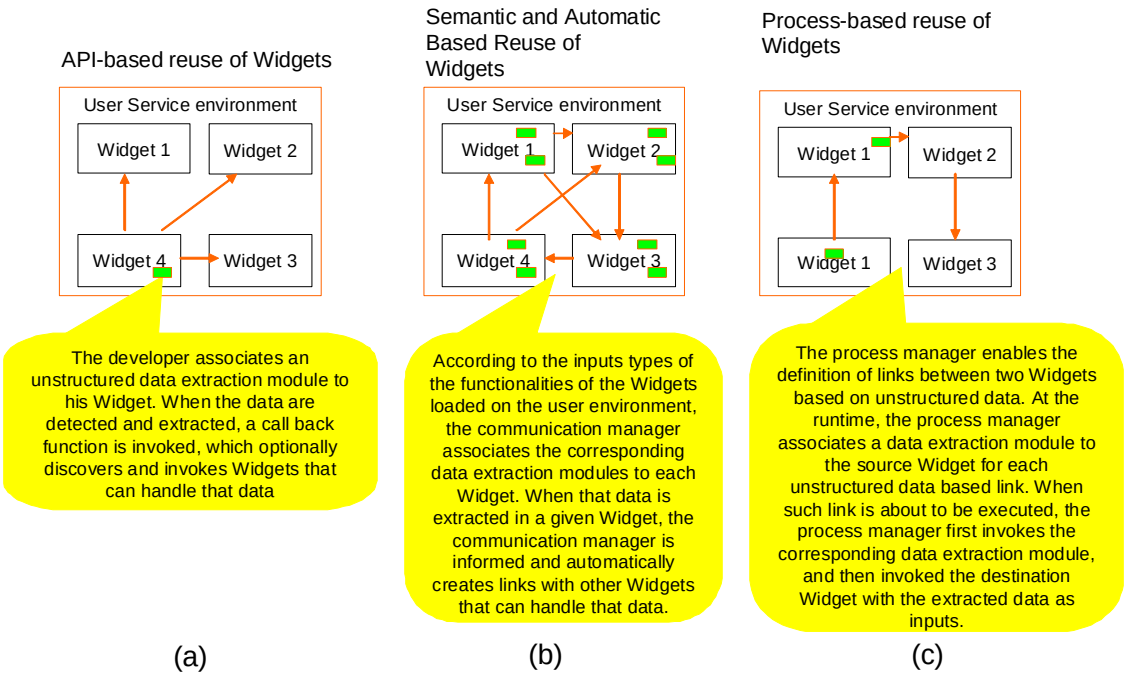


Figure 54: Unstructured data based reuse extension.

The unstructured data based reuse is not exclusive to the WOA. It could be applied to current SOA. However, the Widgets present the advantage of being visual to the users. As a consequence, we harness the intelligence of the users in the process of extracting the unstructured data; the users can check whether the extraction is performed successfully or not.

3.6 Cross-Device Based Reuse Extension

a. Extension Goal

So far, we have considered a user service environment as a Widget aggregator; a Web application that enables the user to access and use several Widgets. The reusability scope is therefore limited to Widgets that are loaded into a single instance of the Widget aggregator, running within a single device. However, users actually have several user service environments; optionally loaded on different devices. This is especially true when considering the proliferation of user devices (PC, laptop, tablet, mobile devices, TV…etc); devices which are more and more sophisticated from one hand, and enable the user to access and use several services from another hand. Current SOA solutions addressed for users do not consider this variety of user environments. For instance, the user can not combine a web email service, loaded on his mobile phone, and a video player service loaded on the TV (this would enables him to read attached movies using the TV video player).

To tackle this limitation, we propose in this section to enable the user to combine Widgets loaded on heterogeneous devices. We propose to extend the reuse scope from a single Widget aggregator into several Widget aggregators loaded on different devices. As illustrated in Figure 55, this would enable the user to make a Widget *A* communicates with a Widget *B* easily, even when they are loaded on different devices. This would enable the user to easily connect a web mail service loaded on his mobile phone, with a video player service loaded on the TV, and/or with a PDF reader service loaded on the laptop.

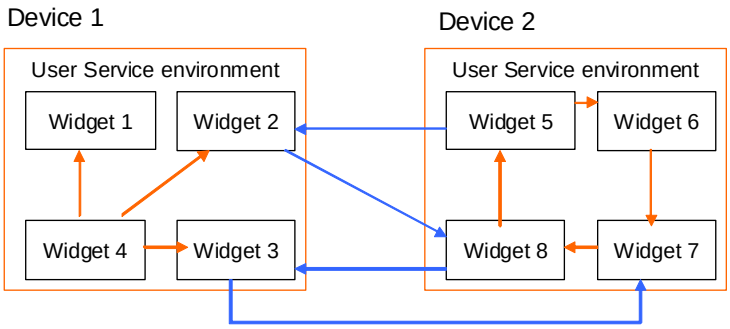


Figure 55: Cross-device based reuse extension goal.

b. *Extension design*

The solution we propose to enable the user to combine Widgets loaded on different devices is based on the definition of a cross-device communication protocol between the different WOA components. This protocol enables different Widget aggregators to communicate with each others to enable the combination of their Widgets. In other words, it is not required to have the same Widget aggregator type on the different devices to enable the Widgets to be combined with each other. The only condition is to comply with the protocol we define. Even web or desktop applications can be combined with the Widgets as long as they are compliant with the protocol we define. As illustrated in Figure 56, this new API represents an extension to the different reuse approaches we have previously defined.

From the technical point of view, this extension is a module used by the three main reuse components introduced in this thesis, namely the API component, the Communication Manager component, and the Process Manager component. The role of this module is characterized by first synchronizing the lists of Widgets capabilities present in the different Widget aggregators instances; and second, when needed, invoking a Widget loaded on a different device. To do that, we define a cross device communication protocol. This would enable different Widget aggregators to communicate with each other, if they are compliant with this protocol. This protocol relies on the Cross Device Communication tool, which provides a communication channel to all instances of Widget aggregators. It enables a module of a Widget aggregator instance to transmit data to another module of another Widget aggregator instance. The

data transmitted between the Cross Device Communication modules (see Figure 56) characterize the protocol we define. Table 10 details the meaning of that data.

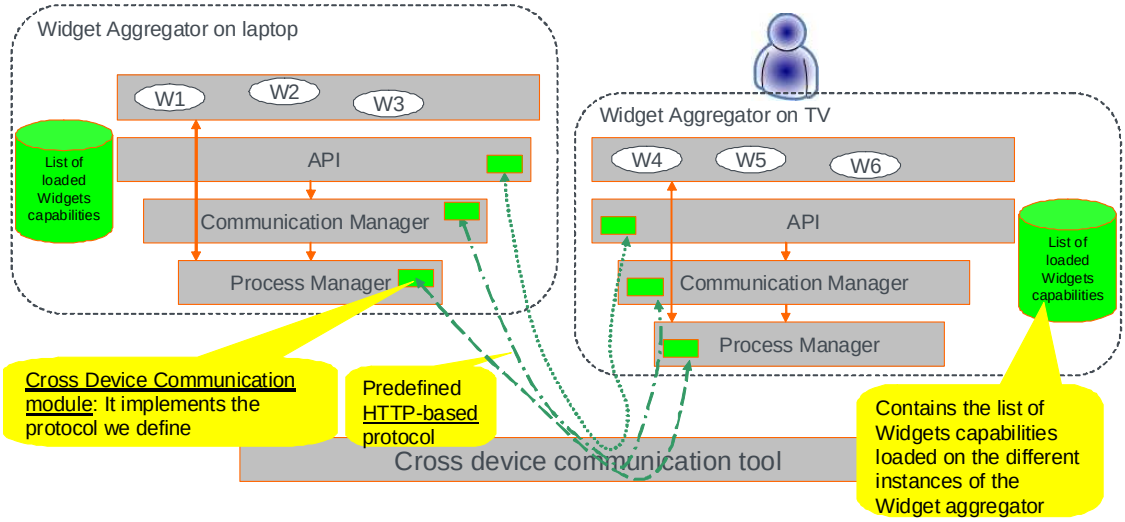


Figure 56: Cross device composition basic architecture.

Table 10. Cross Device Communication Protocol.

Data	**Description**
Registration	This function registers the Widget aggregator instance in the Cross device communication tools, in order to receive information from other Widget aggregator instances loaded in different devices of the same user. Authentication might be required.
Widget_ Subscription_Event	This informs other Widget aggregator instances about the availability of a new capability of a Widget in order to synchronize the list of capabilities.
Publish_Data_Event	This function enables the invocation of a capability of a Widget loaded on another device.
Widget_ Unsubscription_Event	This function informs other Widget aggregator instances about the unavailability of a capability of a Widget in order to synchronize the list of capabilities.

Each time a new Widget capability is available in a Widget aggregator instance, the Cross Device Communication module is invoked to transmit that information using the API (*Widget_Subscription_Event*). Thus, other Cross Device Communication modules, of other Widget aggregators of the same user, receive the information and update the list of Widgets capabilities. Each time a Widget capability is unavailable, the corresponding Cross Device Communication module transmits the information (using the API (*Widget_Unsubscription_Event*)) to the other Cross Device Communication modules, which update the corresponding list of capabilities. These two actions enable each Widget

aggregator instance to have a global view of Widgets that are loaded on different devices and their capabilities.

The protocol we described enables even third party applications, loaded for example on a mobile device, to be combined with Widgets or other applications loaded on different devices. For this purpose, it must just implement the Cross Device Communication module, which registers the application to the Cross Device Communication tool, and maintain a list of Widget capabilities. Then, the application should just read that list to discover capabilities of Widgets that are loaded to the user service environments.

As we are combining Widgets loaded on different devices, it is important to consider the pertinence of making two devices communicate with each other. For instance, it is not pertinent to combine a web mail service loaded on a mobile, and a video player service loaded on a TV, if the two devices are not close with each other (the localization aspect is out of the scope of this thesis). The pertinence of making two devices communicate with each other might depend on other parameters than localization. For example, it might be interesting to combine Widgets of two users belonging to the same social community (group). Thus, the implementation of the mechanisms must be modular enough to easily embed the intelligence that selects which devices should communicate.

Chapter II.3 Widget-Oriented Architecture (WOA) in SOA application fields

In this Chapter, we investigate different SOA application fields within the WOA paradigm. More precisely, we first detail how service composition is performed using WOA; and second, we investigate the business process management using WOA. This Chapter aims to show where the WOA mechanisms we defined succeed compared to SOA, and where they fail.

- Service composition: In Service composition field, WOA is characterized by its user centricity. First, we introduce a static service composition mechanism that enables developers to create Widgets which communicates with other Widgets that are loaded into the same environment. This composition mechanism provides a first and basic user-centric approach for composing services. Second, we introduce automatic and semi-automatic service composition mechanisms using WOA. In each composition approach, we demonstrate how we make use of the abstract service based reuse, the unstructured data based reuse, and the cross device based reuse (mechanisms that are previously defined). The concept of abstract service enables to decouple a composite service (created either using static, automatic or semi-automatic composition mechanism) from the used basic services. This is performed by dynamically binding the functional need of the requestor to a concrete Widget. The unstructured data based reuse introduces a new composition pattern that enables the service composer to capture data that are not declared as legacy outputs of a service, and use them as input parameters of another service. Finally, cross device based reuse enables a user to compose services loaded on different devices. This enables the user to create a pervasive application where functionalities run on the most suited (preferred) device.
- Business process management: The WOA also affects the business process management field. Here in this thesis, we highlight three implications. First, as users can create their own software feature, it is possible for them to define and automate their own business processes. Second, the abstract service based composition, enables a dynamic adaptation of business processes; an adaptation which can be performed for instance according to the context the user. Third, the unstructured data based service composition enables the definition and implementation of business

processes according to unstructured data; data that are neither declared nor formatted when publishing the corresponding service. This is especially useful to capture the data that circulate between employees using communication services such as email, IM, Wiki, Office document…etc.

1 Service Composition using WOA

In the state of the art Chapter, we categorized current service composition approaches into static, semi-automatic, and automatic. Table 11 summarizes the advantages and limitations of each approach when using SOA technologies (details are presented in *Chapter I.1 State of the Art*). For each composition approach, we study in this section the solutions that could be brought by WOA to the limitations that exist within SOA context.

Table 11. Limitations of Current Service Composition approaches

Item		Advantages	Limitations
Service composition	Static service composition	• Services are completely independent from each other. The created service might be as sophisticated as an ordinary application (No technical limitation).	• The creation process is complex. It is conceived only for developers. As a consequence, a long TTM for personalizing an existing service, as well as for creating a new service is noticed. • The created service is tightly coupled to the used basic services.
	Semi-automatic service composition	• Designed for advanced users. It enables personalization. • The time to market (TTM) is low when a user is able to create a service (He is an advanced user). • The created service matches exactly the user needs.	• Not designed for ordinary users; the tools are too complex for them. This implies a long TTM when an ordinary user wants to create a new service, or to personalize an existing one. • The created service is limited. • The created services can not be distributed over the user devices. • The created service is tightly coupled to the used basic services
	Automatic service composition	• Addressed for ordinary users. • It enables a quick creation of a service. • It is very intuitive.	• The created service can hardly match exactly the user needs. • The services are tightly coupled as they rely on a common semantic. • The created services cannot be

			distributed over the user devices. • The created service is tightly coupled to the used basic services

1.1 Static Composition

The main limitations of static service composition within SOA paradigm are summarized as follows:

- Users can not create new services (this is due to the definition itself of static service composition).
- Users can not personalize an existing composite service (a minor change requires a whole development process performed by developers, which implies a long TTM).
- The created service is tightly coupled to the basic services it uses.

After specifying how static service composition is performed using the WOA, we will detail how personalization and loose-coupling are enabled.

The method we introduce to perform static composition of services (Widgets) is characterized by using the API provided by Widget Combination component. More precisely, we use three functions: *GetWidgetList*, *Subscribe*, and *Publish*, which are proposed in the previous Chapter. Figure 57 illustrates a typical sequence, performed by users and Widgets developers, for composing Widgets. First, the user loads different Widgets into his environment (step 1). Second, Widgets (implicitly or explicitly) declare their capabilities using the *Subscribe* function (step 2). Third, Widgets discover the capabilities of other Widgets loaded within the same user environment (step 3). Fourth, Widgets use one or several of the discovered capabilities (step 4 and 5).

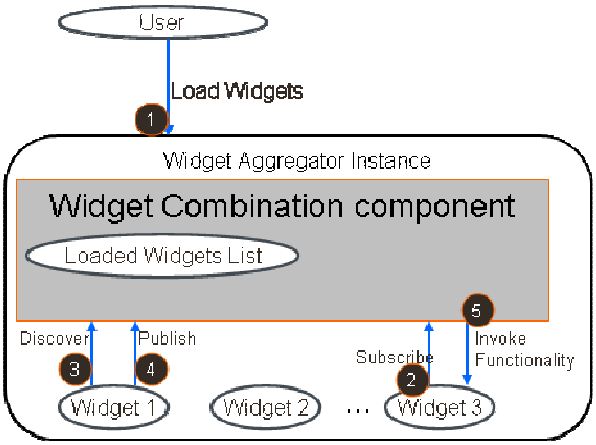


Figure 57: Widget composition using the Widget Combination API

In addition to these three functions, the developers can use the unstructured data based reuse extension, to automatically capture unstructured data within his Widget and publish them to other Widgets.

a. Personalization

WOA provides two approaches for users to personalize a composite service created using a static composition solution. The first approach is manual and the second one is automatic.

As static composition is performed only between Widgets that are loaded on the user environment, users can personalize it by choosing the Widgets to load. For example, if a directory Widget generates and publishes a contact card (which contains first name, last name, and contact information such as phone number, email address, and postal address) to other Widgets, the user can control which Widgets will receive the information and react accordingly. For example, they can load a Telephony Widget to receive the phone number and make calls; and they can also load a Map Widget in order to locate the contact postal address. This personalization approach is the manual one. The personalization scope of this approach is limited to what is expected by the developers of the Widgets. For instance, if the developer of the directory Widget decides to publish only a phone number of a contact, the user can not compose the Widget with a Map Widget (because in this case the postal address is needed).

The second approach for personalizing composite services is automatic and relies on the concept of abstract Widget. This is characterized by loading abstract Widgets into the user environment. The developer can then discover that Widgets and invoke them. The actual Widget to be executed is selected at runtime according to selection rules specified by the user himself. Let's consider, for example, that the user has loaded a "make call" abstract Widget and a directory Widget. Let's consider also that the directory Widget uses the API-based reuse mechanism; it first discovers the Widgets that are loaded by the user, and then optionally reuse their capabilities. In our case, it discovers the "make call" abstract Widget. Thus, when this Widget is invoked, the best "make call" concrete Widget is invoked. The "best" is defined according to the user owns criteria; these criteria may refer for example to the user context such as location and presence, the preferences, the service price…etc.

Though this personalization presents the advantage of being automatic and context aware, it presents the limitation of being static as the user can not create compositions which are not anticipated by the developer of the Widgets. In our example for instance, the user can not compose the directory Widget with a Map Widget, if the developer of the directory Widget does not publish the addresses that will be generated.

b. Loose coupling

In order to decouple composite services from the used basic services, developers discover the capabilities of the abstract Widgets loaded by the users to his environment. The actual Widget being executed is then selected at runtime according to the needed functionality and a set of selection rules specified by the users.

Thus, instead of invoking a specific Widget in a composition, developers invoke abstract Widgets. This mechanism ensures a complete independence between the providers of Widgets and the developers of composite services.

1.2 Semi-automatic Composition

The main limitations of semi-automatic service composition within SOA context are summarized as follow:

- Semi-automatic service composition does not address ordinary users; the tools are too complicated for them. This implies a long TTM when an ordinary user wants to create a new service, or to personalize an existing one.
- The created service is limited (hardly avoidable as this composition approach tries to target users).
- The created services can not be distributed over the user devices.
- The created service is tightly coupled to the basic services it uses.

In this section, we show how we overcome the first, the third, and the fourth issue. We first show how user service composition is enabled. Second, we show how do users create composite services that are distributed over multiple devices. Finally, we show how a created composite service is decoupled from the basic services it uses.

a. User service composition

To enable the users to create by themselves composite services, we combine here two WOA mechanisms: the *"Semantic and Automatic Based Reuse of Widgets"* and the "*Process-based Reuse of Widgets"* defined in the previous Chapter. By this combination, we seek to simplify as much as possible the process of composing services, while providing to users a full control on the defined composite service (based on flowchart definition).

In order to facilitate the process of defining a composite service (constructing the composite service description), we propose to first use the *"Semantic and Automatic Based Reuse of Widgets"* to construct all possible links between Widgets that are loaded by the user into his environment. This generates a "mesh process"; a process which connects all connectable Widgets. Second, we enable the user to delete and automate links. This is possible because each link is visually represented at the frontend by UI element. Each UI element enables the user to activate the link (execute it); automate it, or delete it. Thus, the process of mapping an output of a Widget to an input of another is performed automatically, and the user still can delete undesired links, and/or automate others. The user can also delete a Widget from his environment, in which case the corresponding links are deleted as well. The remaining links construct the process (flow chart) on which the *"Process-based Reuse of Widgets"* relies during the execution of the

composite service. It is important to notice that at the runtime, the user still can change the definition. Figure 58 is a typical sequence diagram. It illustrates the different steps and messages exchange that occur between the components of the architecture.

The most important differentiations of this semi automatic composition using Widget compared to SOA are:

- its implementation at the user service environment as a native functionality,
- the fact that the users do not have to understand the concepts of input, output, and input/output mapping, to compose services,
- and the fact that it is based on the UI of a service instead of XML based formats such as SOAP and WSDL.

Enabling users to compose services is the core differentiation of WOA from traditional SOA technologies. This composition approach is of course implemented and validated in the next Chapter.

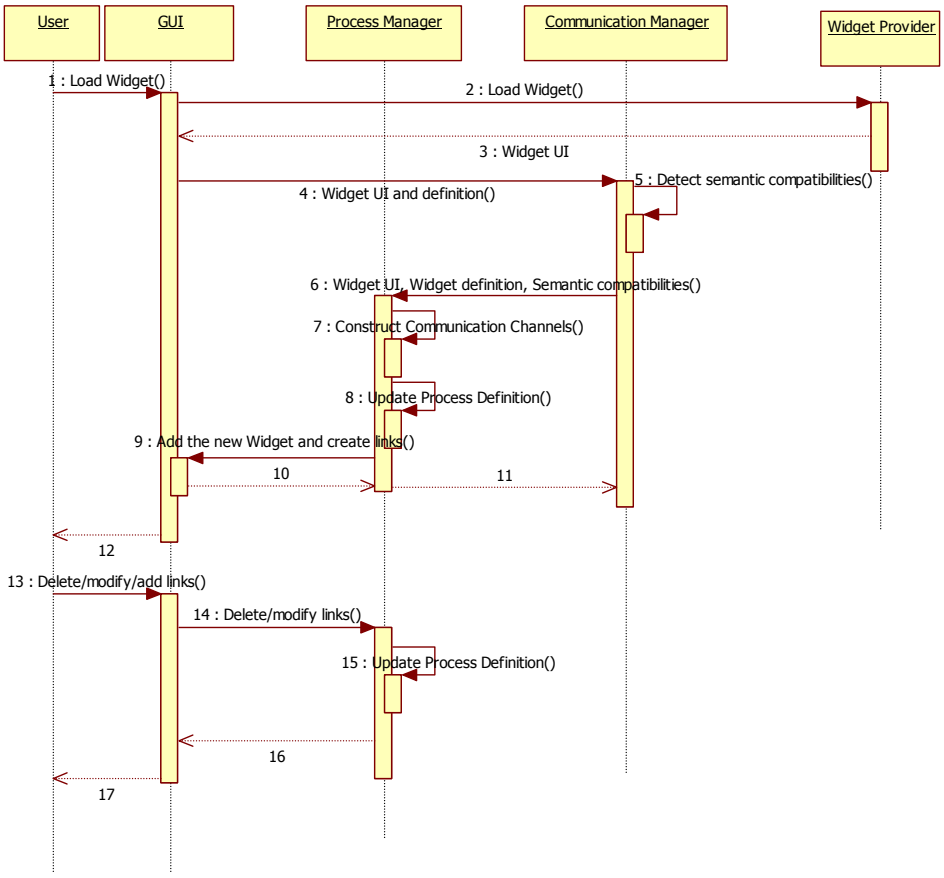


Figure 58: Composite Service creation through Process Manager.

b. Cross-device composition

The cross device based reuse we proposed in the previous Chapter extends semi-automatic service composition from one device to multiple devices. Thus, each time a user loads a Widget into his device, the Widget combination component of that device (implicitly or explicitly) captures its capabilities and publishes them into other Widget combination components of other devices. If a semantic matching is detected the Widget Combination component creates a link between them. This creates all possible links between Widgets loaded on different devices of the same user. It defines a "mesh process". The created links can be automated or deleted by the user. This is possible because each link is visually represented at the frontend by a UI element. Each UI element enables the user to activate the link (execute it), automate it, or delete it. Figure 59 shows a sequence diagram that illustrates the creation of cross device composite service. More precisely, it first shows how two Widgets loaded on two devices are connected with each other to form a composite service; and second, how does the user personalize this composite service.

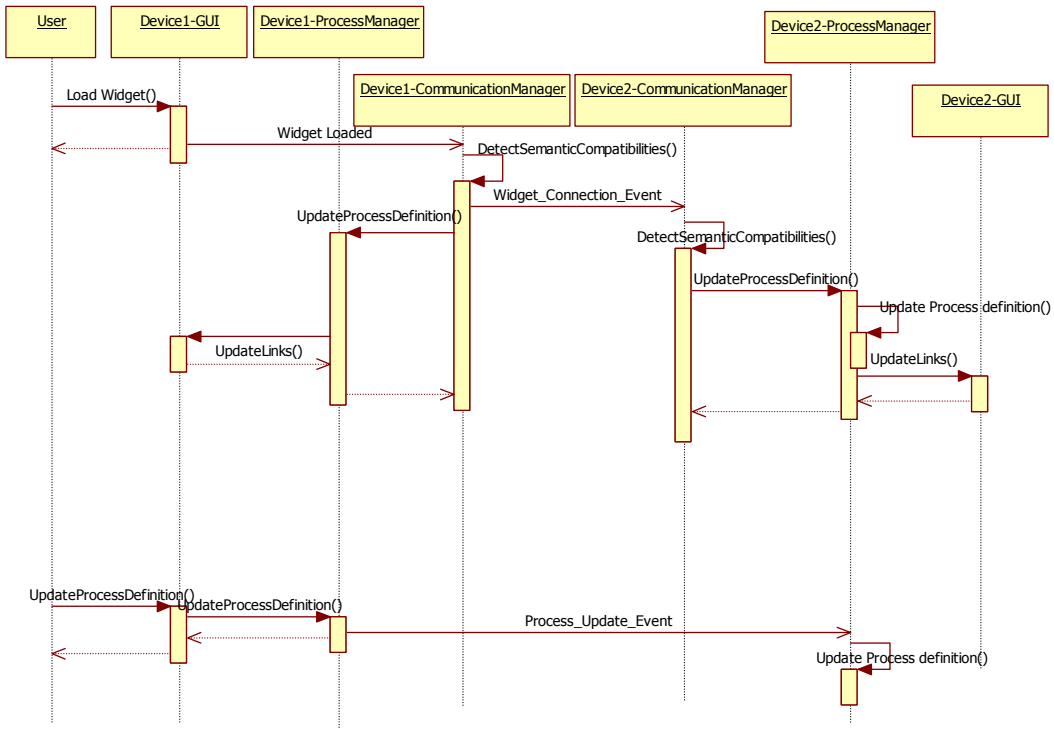


Figure 59: Cross-device composite service creation.

c. Unstructured data based composition

Current service composition tools enables the chaining of services based on structured data; data that are declared as legacy outputs of a service, and formatted according to their type. By contrast, the data that are not declared as legacy outputs of services, and thus not formatted, can hardly be considered within the

composition. That data include for example postal addresses within emails, phone numbers exchanged within an IM conversation, and dates within any document.

The WOA introduces a new feature to tackle this limitation: the composition based on unstructured data. Basically, it is characterized by the introduction of a new pattern in the definition of a composite service. A composite service is thus defined as a Graph *G<N, L, U>* (instead of *G<N, L>*); where *U* is a set of unstructured data based links between services. Each link in *U* is defined by quintuplet *c (Source-Widget, Output-Type, Destination-Widget, Destination-Functionality, Input-Type)*, where *Source-Widget* – an element on *N* – is the source Widget of the link; *Output-Type* is the type of the data that should be extracted from *Source-Widget* (this implicitly refers to the data extraction module to use to extract that data); *Destination-Widget* – an element of *N* – is the destination Widget; *Destination-Functionality* is the functionality of *Destination-Widget* which should be invoked; and finally, *Input-Type* is the input parameter of *Destination-Functionality*. Figure 60 illustrates an example of a composite service that includes unstructured data based links.

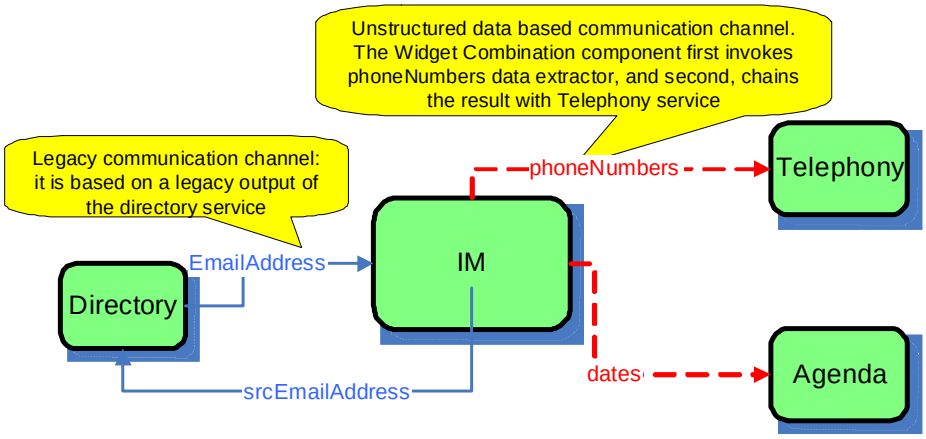


Figure 60: Unstructured data based composite service definition.

At the execution time, when an unstructured data based link is about to be executed, the framework first invokes the corresponding data extraction module, and then invokes the destination service with the extracted data as input parameters.

d. Loose coupling

Semi-automatic service composition in the WOA provides two approaches for performing loose coupling between a composite service and the basic services it uses: the abstract service based reuse mechanism, and a manual adaptation by creating a new composite service (by users).

To use the abstract service based reuse within semi-automatic service composition, performed within WOA, we define the concept of abstract composite script; a flow chart definition that refers to

abstract Widgets instead of concrete Widgets. Thus, each time a node is invoked within the composite service, the framework selects the best available concrete Widget to execute, leveraging the constraint and objective rules associated to the abstract Widget (node). As a consequence, composite services are not coupled to the basic services they use.

The second approach to decouple composite services from the used basic services is characterized by a manual redefinition of the flowchart (composite script). Indeed, as users are now able to compose their own services at the runtime, they can easily modify a composite service. As a consequence, when a basic service is modified, or deleted from the registry, the user can easily replace it by another service that provides the same functionalities, and link it to the other Widgets in order to reconstruct the initial composite service.

1.3 Automatic Composition

As we defined in *Chapter I.1 State of the Art*, an automatic service composition mechanism creates automatically a composite service description from a natural language based request. We have highlighted the limitations of such approach, namely:

- The tight-coupling between composite services and the used basic services (This is resolved exactly in the same way we resolved the tight coupling in the semi-automatic composition of services within WOA).
- The tight-coupling between different service providers (they must use a common semantic; this is hardly avoidable as it constitute the basis of the automatic composition of services).
- The created services can not be distributed over the user devices.
- The difficulty to match exactly the user needs.

Here in this section we propose to tackle the fourth limitation using the WOA. Our proposal is characterized by using the semi-automatic composition mechanism as a failure recovery system. In other words, from a natural language request made by the user from his service environment, the system invokes a Natural Language Composer (NLC) – which generates a composite service definition. From this composite service definition, the Process Manager component generates Widgets within the user service environment; Widgets which are linked with each others. The links are created according to the composite service definition generated by NLC. At this step, the composite service might be not conform to the user needs, but the Process Manager component enables the user to modify it as easily as he creates a composite service (detailed in the previous section). Figure 61 is a sequence diagram that summarizes NLC failure recovery using Widget-oriented architecture.

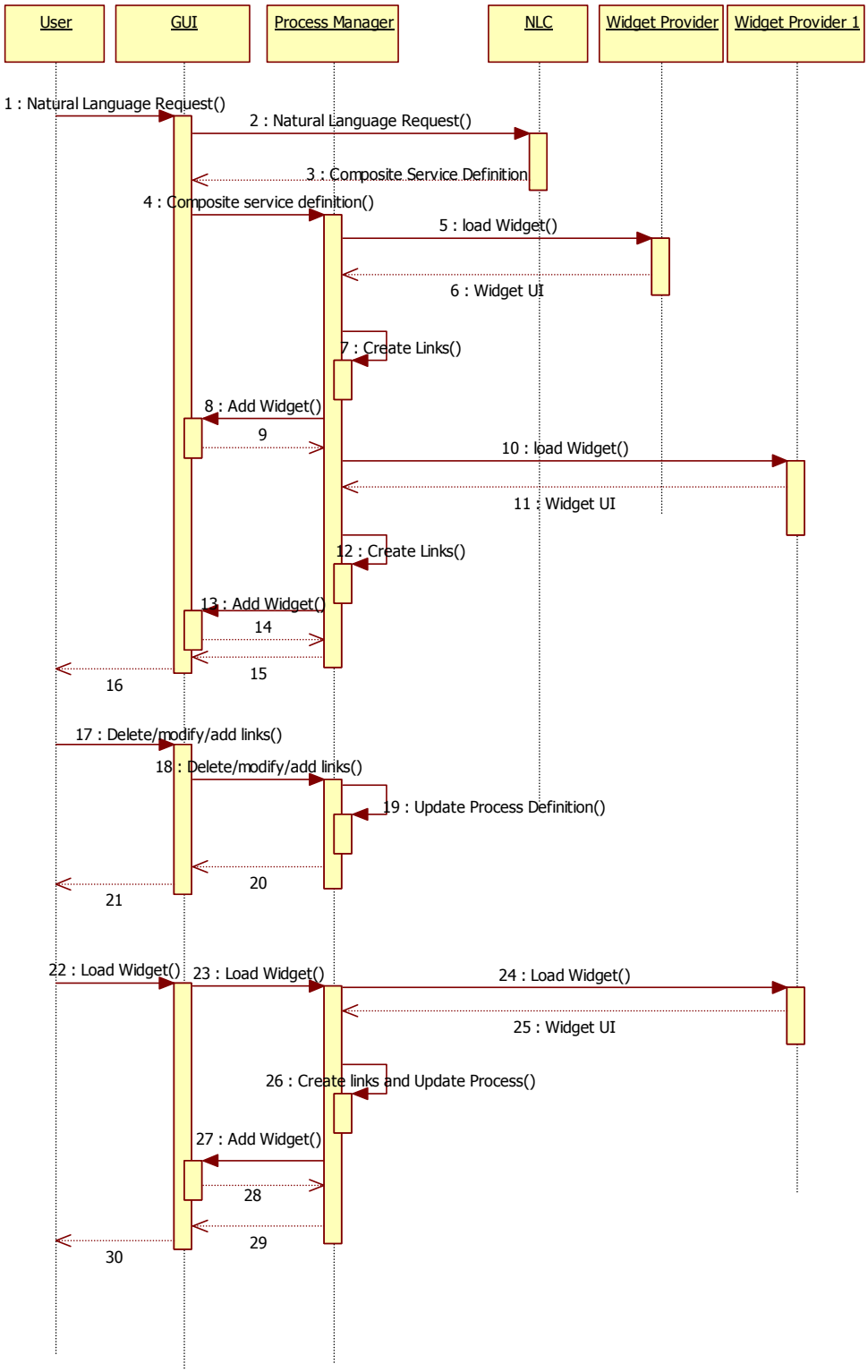


Figure 61: NLC Failure recovery.

2 Business Process Management using WOA

Business Process Management is one field where SOA has succeeded. It enables exposing software applications as Web services. As a consequence, it ensures cross network and cross organization seamless integration. In addition, when using composition tools, professionals can easily model and automate a business process as a sequence of activities exposed as Web services.

However, in *Chapter I.1 State of the Art* we have clearly demonstrated that SOA still suffers from some limitations regarding business process management (BPM). Table 12 summaries of the advantages and the limitations of SOA in this field:

Table 12. SOA advantages and limitations regarding BPM.

Item	Advantages	Limitations
Business process management	• SOA enables a seamless integration of enterprise business processes. It hides the implementation aspects of enterprise applications. • Graphical tools such as BPEL significantly speed up the development of business processes (but it still performed by developers).	• Business processes are heterogeneous, and thus it is hard to capture and implement all the details. In other words, business processes are generalized for the sake of simplicity. • Adaptation to new processes is long as it requires first the capturing of the need, and second its development (usually by a different entity). • The created services can not be distributed over the user devices. • The business process integrator is tightly coupled to the Web Service they use. • Unstructured data are not captured by business process integrators

In the following subsection we detail how does the WOA tackles the listed SOA limitations in business process management.

2.1 Heterogeneity of business processes

As in SOA business processes are specified by business entities and implemented by developers, it is required to generalize the business processes to automate them. As a consequence, several operations are handled manually by the users. This is illustrated in details in *Chapter I.1 State of the Art*.

The WOA enables the users to create their own composite service, through automatic and semi-automatic composition of Widget (see section *1 Service Composition using WOA*). This enables the user to implement himself the business processes he needs. However, in an enterprise context, some actions are common to all users. In addition, it is common in business process modelling to define some actions that are mandatory to achieve a business goal; actions that must be performed by all users to correctly achieve a business goal. As a consequence, it is hardly conceivable for an enterprise to let the employees to define themselves the business processes.

To tackle this limitation we propose in this section to define a business process as a combination of a common part and a user-dependent part. In addition of being common to all users, the common part is also static (does not change frequently within the enterprise). The user dependent part is however heterogeneous (user dependent), and dynamic. For instance, consider the vacation request business process of a team manager, and a purchasing and logistic responsible (see *Chapter I.1 State of the Art*). As illustrated in Figure 62, this business process has a part which is common and mandatory to all users, and another part which is user dependent. The former consists of the vacation request creation, the vacation request study and decision (by the requestor manager), and the response notification. The latter is depends on the user. The team manager for instance would update the agenda, send email to the team, and sett up an automatic email response during the vacation period; and the purchasing and logistic responsible would search pending purchasing orders, call product providers, redirect incoming calls during the vacation period, and update the agenda .

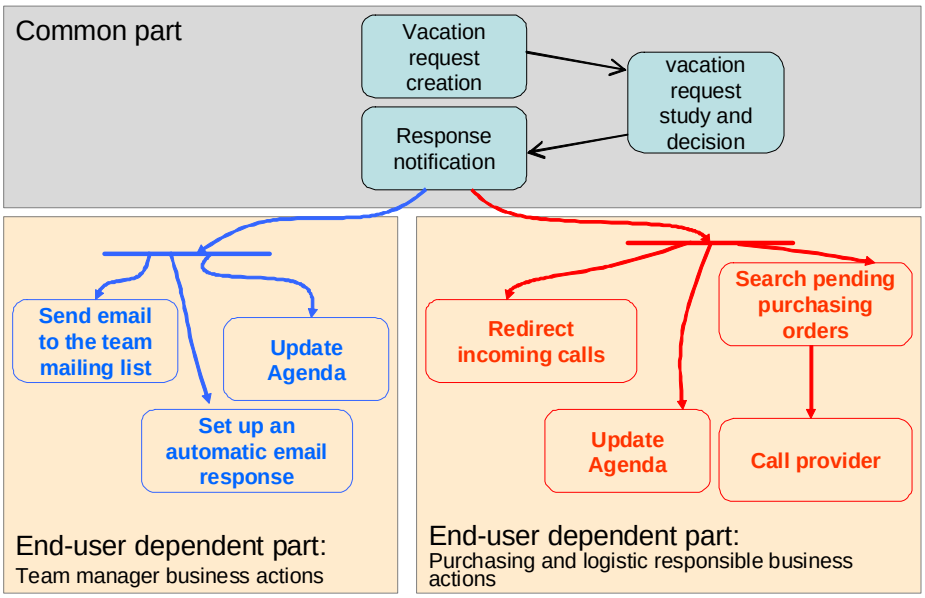


Figure 62: Business process modelling and implementation

After splitting business processes into a common part and a user-dependent part, we propose to combine WOA and SOA to tackle the heterogeneity and dynamicity of business processes, while enabling business process entities to define and/or force some actions to perform successfully a business goal. Our proposal is to combine SOA composition approach and the WOA composition approach. The former is in charge of automating the common part of processes. The common part of processes is thus designed by business entities and implemented by developers using SOA composition technologies. The latter is in charge of automating the user-dependent part of processes. The user-dependent part of processes is thus designed and implemented by the users themselves; using Widget-based composition approaches (automatic and semi-automatic) previously detailed. Figure 63 illustrates the technical aspects of how business processes are automated using our proposal (see Figure 20 for comparison with the traditional approach); and Figure 64 depicts a sequence diagram that illustrates the actions performed by different entities to model and implement a business process (it is interesting to compare it with Figure 22 to see the difference with business process adaptation using SOA technologies).

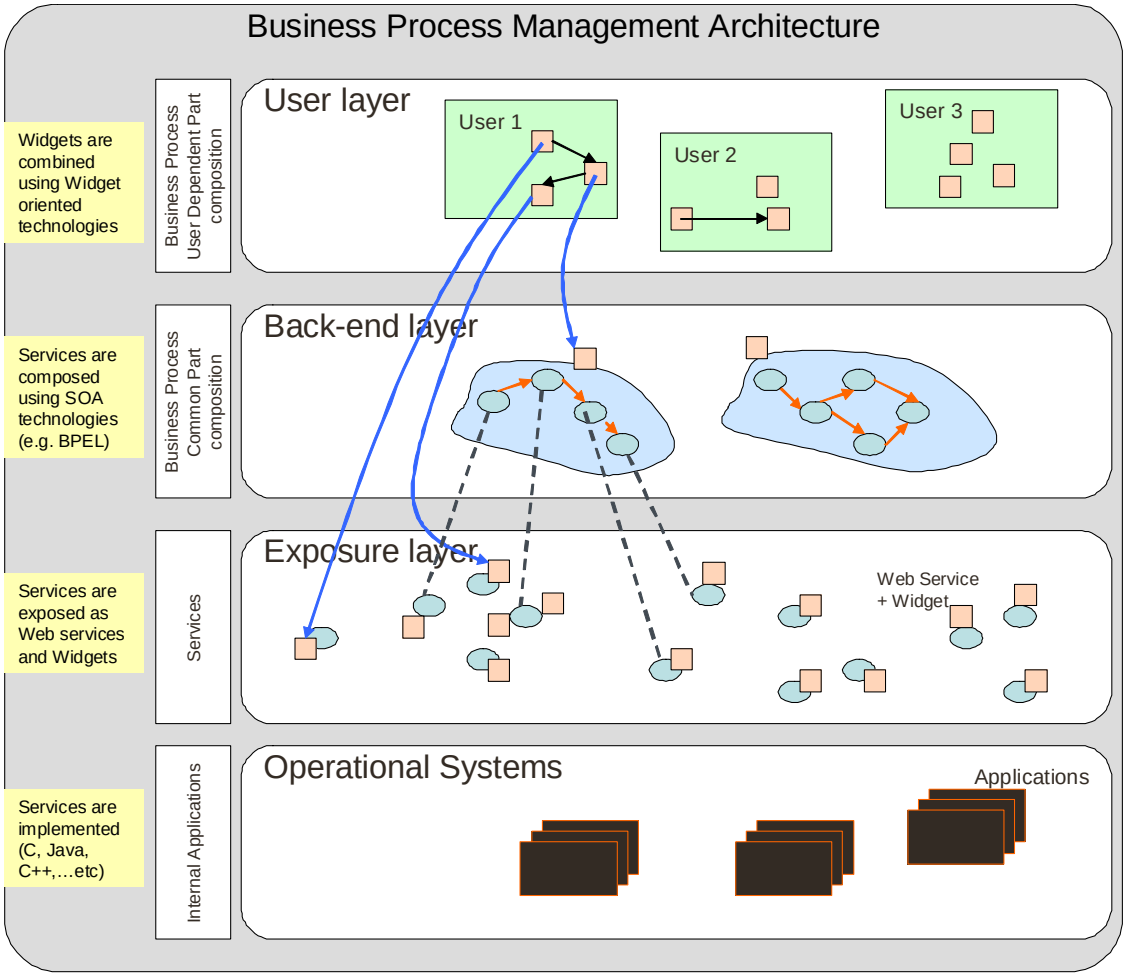


Figure 63: Business process automation proposal.

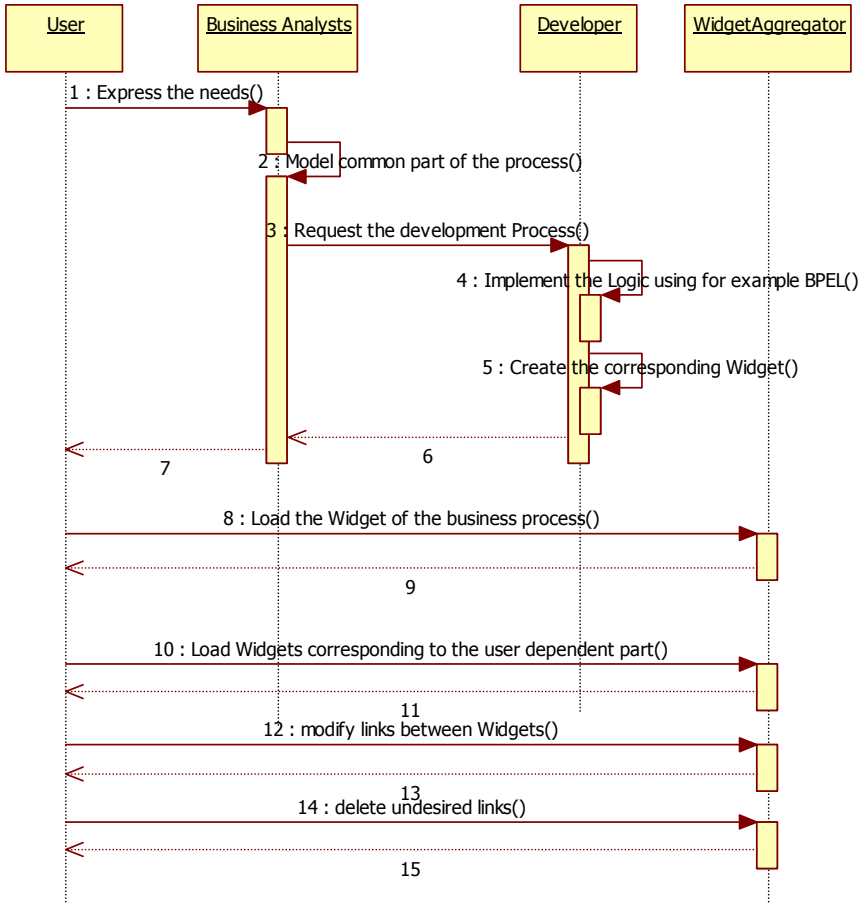


Figure 64: Business process creation using Widget-oriented architecture.

This section has tackled the issue of modelling heterogeneous business processes. The next section will deal with their dynamicity. Notice that the dynamicity is mainly occurring on the user-dependent part of the business process.

2.2 Adaptation of business processes

In order to tackle the dynamicity of business processes, one solution would be a dynamic adaptation mechanism. The dynamic adaptation is an adaptation which is performed when the user needs a new configuration of his business process. Using the mechanisms we have previously defined, we propose in this section two approaches for making such adaptation. The first one is based on the usage of composite service creation capability provided to users, and the second one is based one the usage of the abstract Widget based composition. The following subsections detail respectively the two approaches.

a. Adaptation through abstract Widgets

The first approach we propose to enable dynamic adaptation of business processes is characterized by using the abstract Widget concept as the basis for implementing business process activities. Thus, instead of invoking Web services (in WSA architecture) or Widgets (in the basic Widget architecture), business process developers invoke abstract Widgets as basic elements for performing a given business activity. Let us not forget that an abstract Widget, without referring to any concrete implementation, refers to a functionality associated to a set of selection rules; the selection rules enable to dynamically select the most appropriate concrete Widget to invoke in order to perform a given functionality. The selection rules might be forced by business process analysts, or configurable by the users themselves.

To illustrate that abstract Widget based composition enables the dynamic adaptation, let us consider a basic vacation request business process, which is mainly composed of a vacation request activity, and an update agenda activity. Consider also that the users in the company have a freedom of choosing the agenda service they want to use. As a consequence, the agenda service may differ from one user to another. The abstract Widget concept enables business process developers to implement that activity (agenda update), without referring to any specific concrete service. Eventually, users, or the business process analysts, select the selection rules they want to apply on that abstract Widget. As a consequence, the activity realization is adapted according to the users' choice and context. Figure 65 shows a sequence diagram illustrating the different interactions.

b. Adaptation through personalization

The WOA provides the users with the capability of easily combining Widgets with each other. This capability, associated to the business process modelling approach described in the previous section, enables a dynamic adaptation of business processes. The adaptation is mainly concerned with the user-dependent part of the business process, which is very dynamic.

In order to illustrate how do users use the Widget combination capability to adapt a business process according to new activities, needs, or requirements, we rely on the vacation request business process example. This business process is composed of a common part and a user dependent part as previously detailed. We suppose that the common part is implemented as a single Widget, which enables the user to create a vacation request, send it to his manager, and be notified about the decision. We suppose also that the user dependent part is composed of the following activities: setting up call redirections during the vacation period, updating the agenda, and setting up an automatic email response during the vacation period. Figure 66 illustrates the Widget aggregator environment of the user populated with the

corresponding Widgets. These Widgets are linked with each others according to the user-dependent part of the business process (as detailed in the previous section).

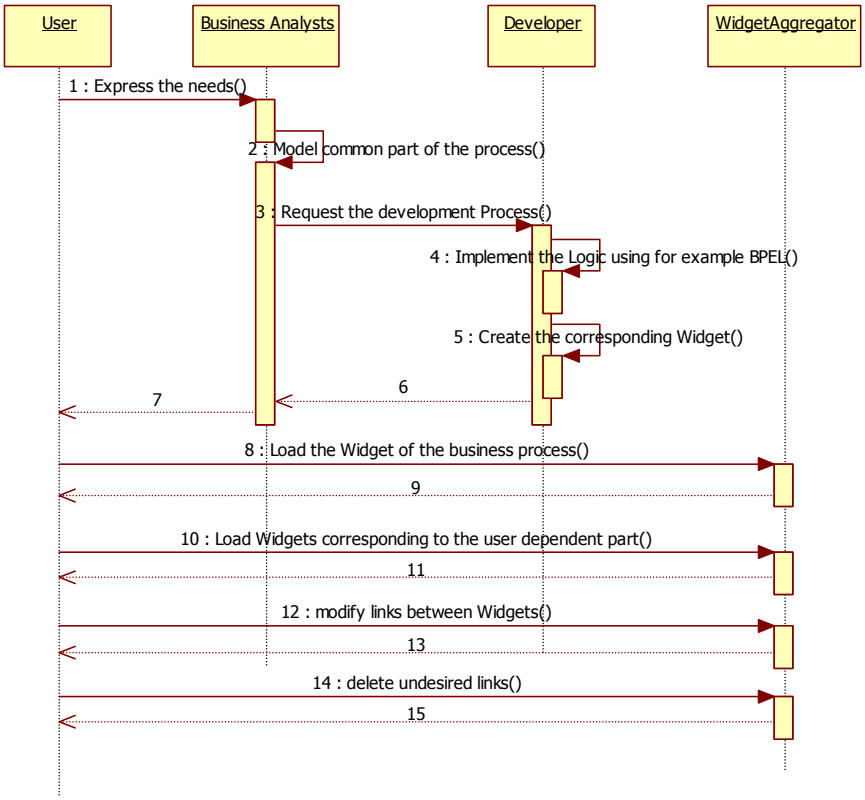


Figure 65: Business process adaptation using abstract Widgets.

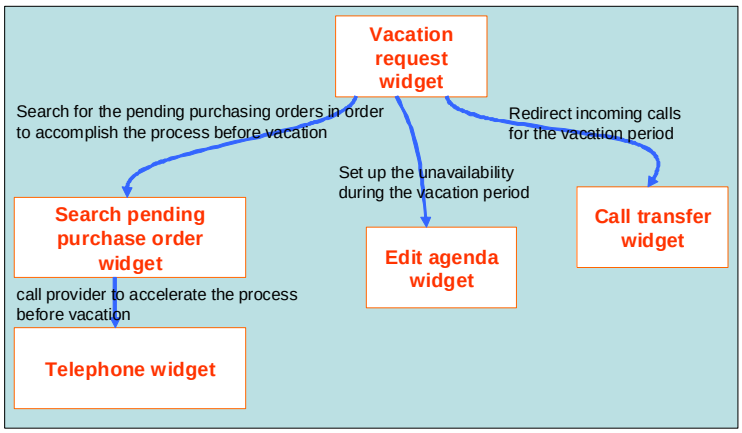


Figure 66: Widget aggregator configured according to a business process.

In order to illustrate the adaptation process, let us assume that:

- The user wants to perform a new activity before leaving (for vacations); this new activity is characterized by sending emails to a set of colleagues to notify them about the vacation period.

- The agenda service has changed; the company has decided to use another agenda service which is, for example, less expensive and more efficient.

Using the Widget combination capability, users can reconfigure themselves the business process user-dependent part, so that they automate their tasks. To configure the new activity for example (to respond to the first assumption), they have just to load the corresponding Widget; in our case, the sending email Widget. This action, as detailed in the semi-automatic composition, implies the automatic linkage of that Widget with all other compatible Widgets. Optionally, the user may then delete undesired links.

Concerning the second assumption, to configure the new agenda service, the user should just delete the Widget of the older agenda service, and replace it with the Widget of the new agenda service. The links between the new agenda Widget with the other Widgets that are present within the user environment are automatically created. Optionally, the user should then delete undesired links. Figure 67 depicts a typical sequence diagram that shows the adaptation process; neither business analysts nor developers are involved within the process. It is interesting to compare it with Figure 22 (*Chapter I.1 State of the Art*) to see the difference with business process adaptation using SOA technologies.

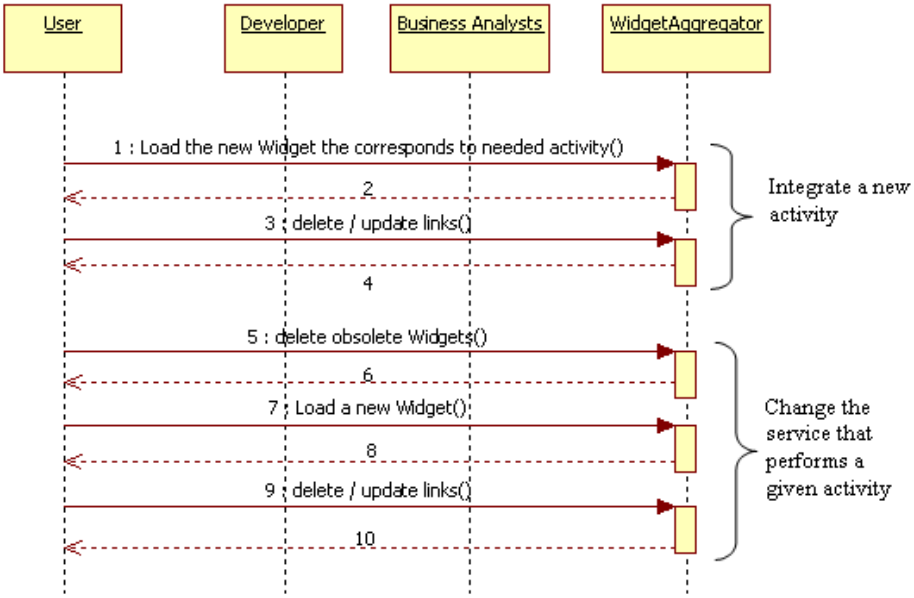


Figure 67: Business process adaptation sequence diagram.

This approach for adapting business processes has the advantage of being more general than the abstract Widget based adaptation. Unlike abstract Widget based adaptation which enables only the modification of the services that perform a given activity, this approach enables the modification of the logic of the business process. It enables:

- modifying the execution sequence of activities,

- integrating new activities, and removing others,
- and changing the Widgets that perform a given activity.

However, compared to the abstract service based adaptation, this approach present the limitation of being manual and applicable only for the user dependent part.

2.3 Loose coupling between integrators and basic service providers

One limitation of SOA technologies regarding the business process implementation is the tight-coupling between the implementation of the business process and the basic services it uses. It is especially due to the fact that the activities of the business processes refer to concrete and basic services. This is true even when considering the partitioning of business processes into a common part and a user-dependent part we have proposed.

The abstract Widget based composition enables to tackle such limitation. Indeed, the mapping of an abstract Widget into a concrete Widget is performed dynamically, at runtime. When a composite service (a business process implementation) is defined as a set of chained abstract Widgets (functionalities and a set of selection rules), there are no reference to a concrete basic Widget. As a consequence, the business process implementation is completely independent from the concrete Widgets that will be executed at runtime to perform the basic activities of the business process.

2.4 Unstructured data capture

As previously detailed, current service composition tools consider only well structured data in the composition flowchart. As a consequence, it is almost impossible to automate business processes that are based on the unstructured data, except if developers do it manually. The unstructured data include for example postal addresses within emails, phone numbers exchanged within an IM conversation, and dates within any document. As a consequence, even though business analysts do not care about such technical details and thus can model a business process according to the unstructured data, it is hardly conceivable for business process developers to automate behaviours based on these data. It is hard because first developers should develop themselves the logic that enables the extraction of those data from services such as email and IM; and second, they must manage themselves the data extraction failure and errors.

The unstructured data based reuse mechanism we have previously defined, provides an efficient approach for considering unstructured data within business process management tools. First, business process integrators (developers or users) create unstructured data based composite services in the same way as they create a traditional composite service. Second, by relying on the Widget based composition, the

unstructured data that are automatically extracted (based on the composite service definition), are visually presented to the user so that he can check their correctness.

3 Conclusions

In this Chapter, we made use of the WOA in two SOA application fields: service composition and business process management. Concerning service composition, our main conclusion is that SOA and WOA are complementary. Indeed, while SOA addresses developer needs, WOA is more user centric, and focus on personalization and simple composition addressed for users. Existing semi-automatic composition approaches in SOA provide the advantage of being very flexible as they decouple the definition of service logic from the UI. Consequently, advanced user interfaces could be created. However, ordinary users can hardly use these tools. Indeed, they first still remain based on flowcharts in the definition of the service logic; flowcharts can be hardly understood by ordinary users. Second, the UI creation is either manual in which case ordinary users can not achieve it, or automatic, in which case the result is a basic and not user friendly. The WOA in contrast simplifies significantly the creation of the composite services. However, developers and advanced users can not create sophisticated interfaces using WOA based composition tools we proposed. This is essentially due to our approach of considering the UI as part of the reusable component. Nevertheless, from the ordinary user perspective, this enables him to create much more user friendly interfaces than automatic or semi-automatic tools. In addition, WOA provides an interesting infrastructure for additional functionalities such as unstructured data based composition and cross device composition performed by users.

Concerning business process management, the main challenges are the heterogeneity and dynamicity of business processes. In this Chapter we have first proposed to split each business process into a common part and a user dependent part. The common part is composed of the actions that are common to a significant population of users, and the user dependent part is specific to a given user, which makes it heterogeneous and often dynamic. Second, the common part is automated by developers using SOA and the user dependent part is automated by users themselves using WOA. This combination of SOA and WOA enables business analysts of a given company to tackle the heterogeneity of business processes while having the control on the company processes which are common to all users.

Table 13 shows in more details the solutions we proposed in this Chapter, and the SOA limitation they respond to.

Table 13. WOA solutions to SOA limitations.

SOA/WOA application field		SOA limitations	Resolved (yes/no)	WOA proposed solutions
Service Composition	Static service composition	Long TTM for personalizing an existing service.	Yes	Widget Combination API. A composition mechanism limited to services loaded by users.
		Long TTM for creating a new service.	No	-
		Tight coupling.	Yes	Abstract service based reuse (Interpreter component).
	Semi-automatic composition	Not addressed for ordinary users.	Yes	Widget based composition. It is characterized by capturing (at the front end, at the runtime) the semantic compatibilities of Widgets, creating links between them, and enabling the user to personalize these links.
		Limitation of created services.	No	-
		Cross device composition of services.	Yes	The cross device based reuse of Widgets.
		Tight coupling.	Yes	Abstract service based reuse.
				Modification at runtime by users themselves.
	Automatic composition	Matching user needs.	Yes	Enabling the users to modify/adjust themselves a created service.
		Cross device composition of services.	Yes	The cross device based reuse of Widgets.
		Tight coupling.	Yes	Abstract service based reuse.
				Modification at runtime by users themselves.
		Shared semantic between providers.	No	-
Business Process Management		Business processes are heterogeneous, and thus it is hard to capture and implement all the details.	Yes	Splitting business processes into a user dependent part and a common part. Then, enabling the users to automate themselves the user dependent part.
		Adaptation to dynamic processes.	Yes	Modification at runtime by users themselves.
				Abstract service based reuse for an automatic adaptation.
		Tight coupling.	Yes	Modification at runtime by users themselves.
				Abstract service based reuse.
		Unstructured data are not captured by business process integrators.	Yes	Introduction of a new service composition pattern based on unstructured data.

Part III Implementation and Validation

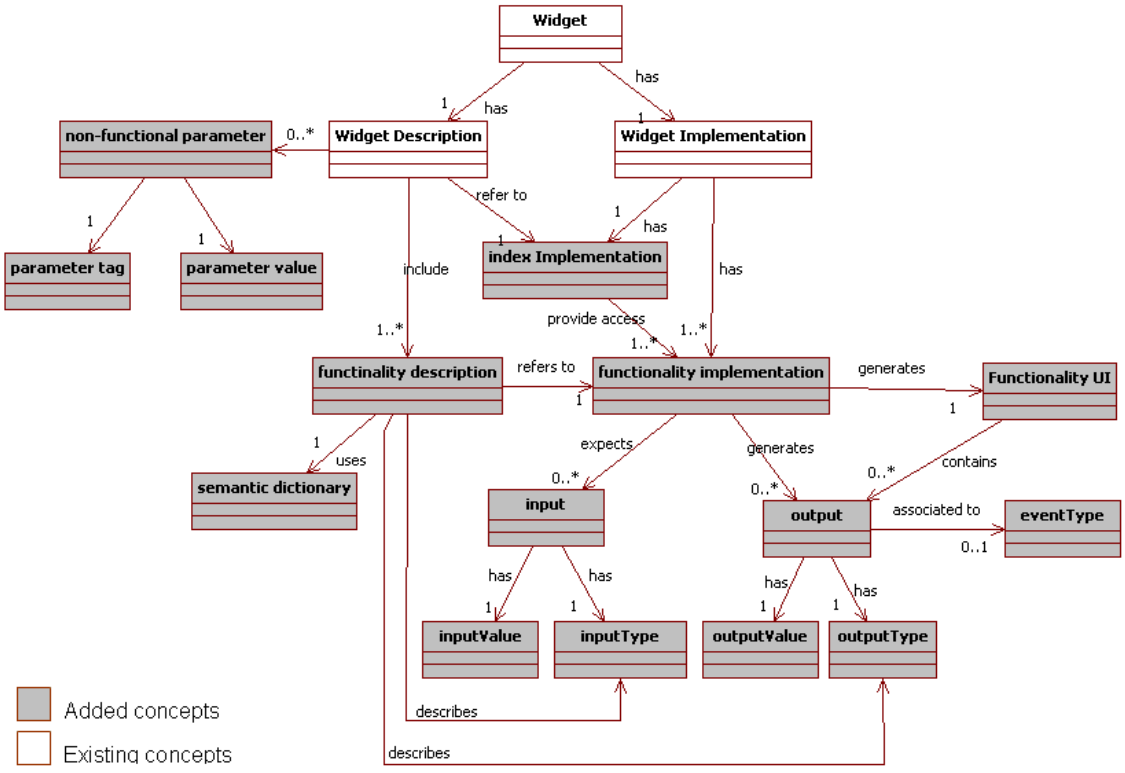
In the previous part, we have defined the WOA principles and proposed a design of an end-to-end architecture. In this part we will detail its implementation aspects and illustrate the result through use cases and screenshots. The first Chapter details the implementation of the WOA, including the different reuse mechanisms we have previously defined (API-based reuse, Semantic automatic based reuse, Process based reuse, cross-device based reuse, abstract service based reuse, and unstructured data based reuse). The second Chapter illustrates, through use cases and screenshots, the usage of the defined and implemented architecture within the two considered SOA application fields, namely service composition and business process management. Finally, the third Chapter details the experimentation and the different demonstrations we made, along with users feedback and different projects that uses or intend to use the WOA paradigm.

Chapter III.1 An Implementation of WOA

This Chapter details the WOA implementation. We first detail the implementation of a Widget. Second, we detail the implementation of the Widget aggregator, and illustrate its basic functionalities. Third, we detail each innovative mechanism we introduced for an efficient Widget reuse.

1 Widget

Figure 40, which is duplicated here, details the different parts of a Widget in our architecture. We will provide the implementation of each part in this section.



Basically, each Widget has a description file and an implementation. The description file is provided by a Widget provider. It is an XML file which can be created manually by the provider of the Widget, or generated automatically after performing a Widget publication process. The publication process is characterized by filling a Web based form in which the Widget provider provides the required information to describe the Widget.

The description file of a Widget contains mainly the index (main page) of the Widget, and the description of each functionality it provides (URL of the functionality, goal, inputs types, and outputs types). Figure 68 is an example of a Widget description file.

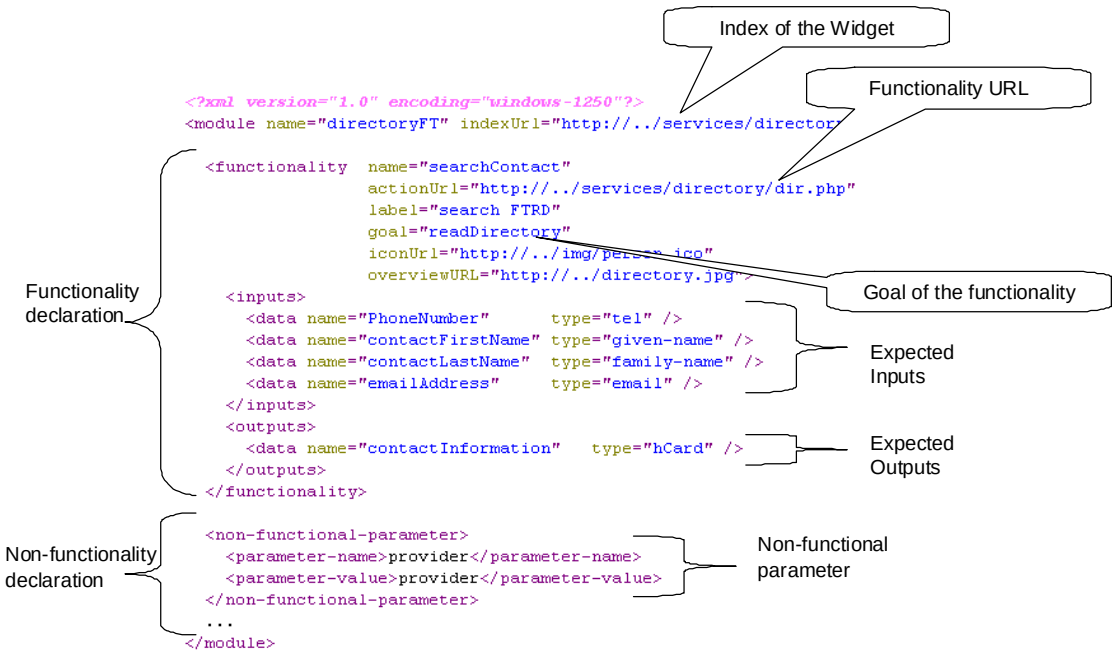


Figure 68: Widget description file.

Associated to this description file is the implementation of the Widget. The Widget implementation must fulfill the following requirements:

- It must be accessible through the index URL specified in the description file.
- It must be Web based. In other words, it must use current Web standards such as XHTML, JavaScript (JS), and CSS.
- Each functionality must be accessible through the URL, using GET or POST method, as specified in the description file.
- The input parameters are passed as GET or POST parameters in the HTTP request, using the parameters names specified in the description as parameters names in the request.
- The outputs of each functionality are annotated in the UI (rendered by the functionality URL), using the name specified in the description file.

Each widget implementation includes different modes (View, Edit, and Help). The different modes are defined within the UI of the Widget with special tags that a Widget aggregator must understand. For example, to add a login/password configuration form to a Widget, the developer must add the following HTML snippet (Figure 69).

```
<form class="configuration">
  <p>
    <label>login</label> <input type="text" name="login" value=""/>
    <label>password</label> <input type="password" name="password"/>
  </p>
</form>
```

Configuration parameters form

Figure 69: HTML snippet for Widget Configuration.

In this thesis we consider that View mode, Configuration mode, and Help mode are accepted by all Widget aggregators. This assumption is supported by the fact that these three modes are present in current two popular Widget standards (JSR168/268, and W3C widget specification). In addition to these three modes, we recommend the integration of a notification mode and a reduced mode. The notification mode enables the Widget to notify the user about the occurrence of an event (e.g. incoming call, message reception…etc). This is especially useful for communication services. The reduced mode is a representation of a Widget in an area that shows that it is running but not fully displayed.

2 Widget aggregator

The implementation of the Widget aggregator we propose is based on XHTML, JavaScript (JS), CSS, and PHP. We have also used (and modified) DOJO[34] JavaScript library to facilitate the management of the Widgets and avoid cross browser issues. The Widget aggregator incorporates a set of components illustrated in Figure 70.

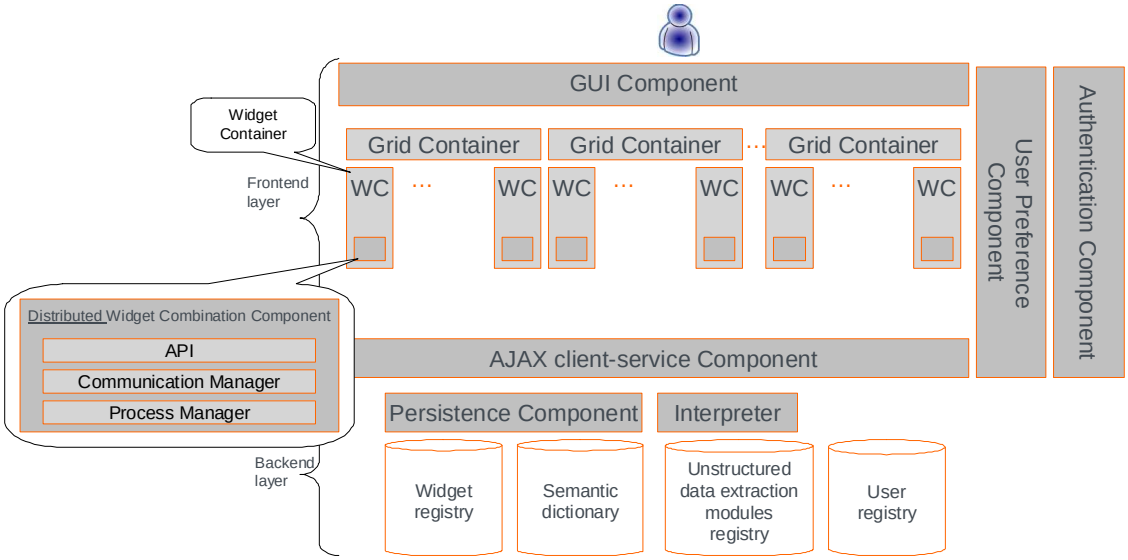


Figure 70: Basic Components of the Proposed Widget Aggregator.

- The GUI component is a Web page that provides the frontend UI. It enables the authentication of the user (through the authentication component), and the personalization of his

[34] DOJO toolkit, http://dojotoolkit.org/, accessed on June 8th, 2010.

environments (by creating new tabs, loading new Widgets…etc.). Figure 71 illustrates the Widget aggregator GUI.

- The Grid Container component creates a drag and drop area in the Web page. In this area, users can dynamically add, remove, and move Widgets. Each tab in the GUI instantiates a grid container object.

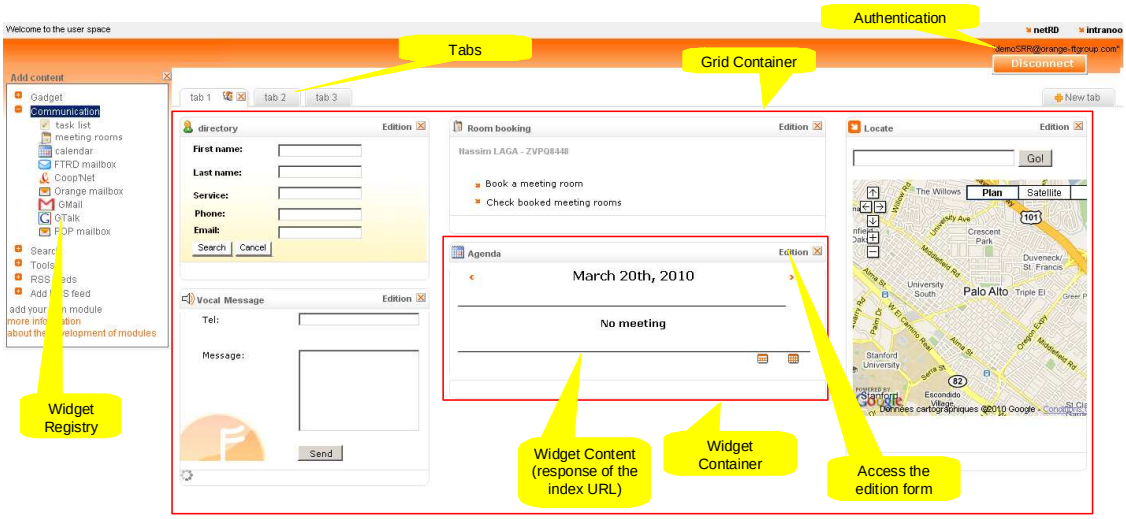


Figure 71: Widget Aggregator Illustration.

- The Widget Container (WC) component is instantiated each time a Widget is loaded to the GUI. Each instance is associated to a Widget. This component is in charge of invoking and executing the Wigdet, and managing its entire lifecycle. It receives as input the Widget description file URL. It extracts the index URL of the Widget. It invokes the Widget. It parses the response (XHTML based) in order to detect special tags such as the configuration form. The Widget Container is implemented as an extension to the Widget object of DOJO library (see http://api.dojotoolkit.org/jsdoc/1.2/dijit._Widget).
- The Widget Combination component is distributed over the WC components. It implements the different reuse mechanisms we previously defined. Its implementation is detailed in the next subsection.
- The Authentication component is in charge of authenticating the user.
- The User Preference component is in charge of saving and loading all user related parameters from the database such as: user preferred widgets, their place in the web page, and their configuration parameters.
- The AJAX client-server component is a JS API based on DOJO, which facilitates the interaction between the frontend components and the backend (server side) components. It facilitates for example the retrieval of the list of existing Widgets (to be displayed for the user

on request), description of a specific Widget, and the user related data (e.g. list of Widgets he loaded on the environment, their places in the environment, the list of tabs he created…etc).

- The Persistence component provides access to the database content (Widget registry, User Registry, The semantic dictionary, and the Unstructured data extraction module registry).
- The Interpreter component is implemented as a Servlet. It is in charge of selecting the best concrete Widget to execute according the needed functionality and a set selection rules.

3 Widget Combination Component functionalities

The Widget Combination component is the main component introduced in this thesis. It performs a user centric combination of Widgets. The implementation of this component is realized at the frontend, except the Interpreter component for the resolution of abstract Widgets (selection of the best available Widget according to a functional need and a set of selection rules) which is implemented at the backend. In the following subsections, we first detail the implementation of the three components that provide the reuse capabilities, namely API, Communication Manager, and Process Manager; then, we detail the implementation of the extensions we defined, namely, Abstract Widget based reuse, Unstructured data based reuse, and cross device reuse.

3.1 API

The API component provides developers with a set of JS functions detailed in Table 14. Basically, these functions enable developers of Widgets to discover at the runtime the Widgets that are loaded at the user service environment, and optionally reuse their capabilities. Similarly, the API enables developers to expose their functionalities to other Widgets that are loaded to the user service environment.

Table 14. API.

Function Name	Parameters	Description
subscribe	*dataType, Goal, urlCallBack, HTMLElement*	This function enables Widget developers to dynamically declare a functionality provided by the Widget. The *dataType* is the type of the expected inputs. The *Goal* is the functional goal description of the functionality. *urlCallBack* is the invocation URL of the functionality. *HTMLElement* is an HTML element (e.g. icon, or text) that enables other Widgets to add a UI element that enables the user to launch a functionality of another Widget.

	eventType, dataType, Goal, urlCallBack	This function enables Widget developers to dynamically declare a functionality provided by the Widget. The invocation of this functionality is conditioned by the occurrence of an event of type "*eventType*".
unsubscribe	*dataType, Goal, urlCallBack*	The Widget declares its incapacity to perform a previously declared functionality. This is useful to change the Widget capabilities according to the state of the Widget (e.g. if a Telephony Widget is "in communication" state, it can not make new calls).
	eventType, Goal, urlCallBack	The Widget declares its incapacity to perform a previously declared functionality (conditioned by the occurrence of an event of type *eventType*).
getWidgetList	*Goal*	Enables Widget developers to retrieve the list of available functionalities (on the user environment) that perform *Goal*. (e.g. *getWidgetList("makeCall")*).
	dataType	Enables Widget developers to retrieve the list of functionalities that are able to receive data of type "*dataType*" as input.
	eventType	Enables Widget developers to retrieve the list of functionalities that can be invoked under the occurrence of *eventType*
publish	*WidgetId, functionality, dataType, dataValue*	Invoke the Widget functionality with *dataValue* as input parameters. If *WidgetId* equals "*", this function invokes all Widgets that provide operations which perform the specified functionality. If *functionality* "*" and *WidgetId* equal "*", this function invokes all functionalities that are able to receive data of type *dataType* as input parameters.
	WidgetId, functionality, eventType, eventValue	Invoke the Widget functionality with *eventValue* as input parameters. If *WidgetId* equals "*", this function invokes all Widgets that provide operations which perform the specified functionality. If *functionality* "*" and *WidgetId* equal "*", this function invokes all functionalities that are

		invoke-able through the occurrence of event of *eventType* condition.

The implementation of this API is distributed over the WC components. Each WC component embeds a JS object, named *JS_API*, that provides the listed functions. Each *JS_API* maintains a list of the capabilities of the Widgets loaded on the user service environment. The different *JS_APIs* auto-discover each others, and exchange the information related to the capabilities of the Widgets in order to update and synchronize their lists. Each time a function of the API is invoked by the Widget logic, this information is transmitted to the related Widgets. For example, when a *subscribe* function is invoked on a Widget *A*, this information is transmitted to other *JS_API* object of other Widgets loaded within the same environment. Figure 72 illustrates a typical sequence of calls to JS functions. After discovering the *JS_API* objects present in the user service environment (step 1), Widget 1 invokes the subscribe function to declare its capabilities (step 2). The *JS_API* updates its capabilities list and transmits the information to other *JS_APIs* of other Widgets present in the user service environment (step 3). Each *JS_API* object which receives such information (availability of new capabilities) updates its capabilities list. When, Widget 2 invokes the publish function (step 4), the *JS_API* detects which Widget is concerned with this publication and invokes the corresponding capability (step 5 and 6).

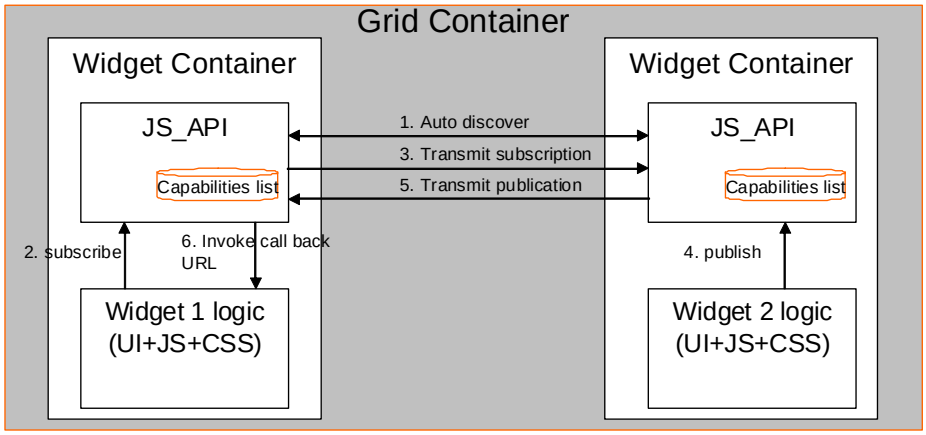


Figure 72: Widget combination API Distributed mechanism.

Figure 73 is a screenshot that shows a Telephony Widget that uses the capabilities provided by a directory Widget (Search a contact from a phone number). When an incoming call occurs at the telephony Widget, the Widget first discovers the Widgets that are present in the Web page and accept phone number as input parameter (*getWidgetList(tel)*), and then it invokes them using *publish* function.

Figure 73: Illustration of Widget reuse through Widget Combination API.

3.2 Communication Manager

The semantic and automatic based reuse of Widgets is a mechanism that enables the reuse of Widgets by the users themselves. It is characterized by an automatic detection of semantic matching between functionalities loaded at the user service environment, and an automatic creation of links between them. From the user perspectives, loading Widgets into the same grid container (tab) is sufficient to combine them.

From the technical perspectives, this mechanism is implemented by the Communication Manager component; a JS object (*JS_CM*) distributed over the WCs. The *JS_CMs* exchange the different information between them exactly in the same way as illustrated in the previous subsection. This component manages the Widget lifecycle, and automates subscriptions and publications between compatible Widgets (compatible Widgets are detected based on their descriptions). The Widget lifecycle includes mainly: the Widget connection phase, the Widget running phase, and the Widget disconnection phase. The management of the lifecycle of Widgets is important to keep the created links between them coherent. In this subsection we will detail the actions performed by the *JS_CM objects* during these different phases.

To illustrate the actions performed during the different phases (by the *JS_CMs* that implement the mechanism), we suppose, as an initial state, that the user has already loaded into his environment a directory Widget. This Widget embeds one functionality: search through contact information (e.g. first and last name, phone number, and email address…etc.). It generates a contact card, which contains complete contact information of a person. Figure 74 illustrates the execution of this Widget.

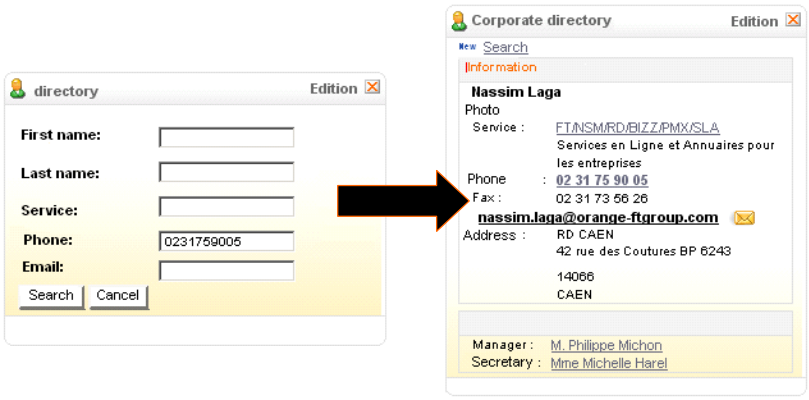


Figure 74: Directory Widget Execution.

a. Widget Connection Phase

To illustrate the connection phase of a Widget, let us consider that the user has requested to load a telephony Widget. This Widget has two functionalities. First, it enables users to make calls by giving a phone number as an input parameter. This functionality generates events such as *ringing*, *communication established*, and *hang up*. Second, the Widget enables the user to receive calls. This functionality, without inputs, generates events such as *incoming call*, and *incoming call accepted*. Associated to these events is a phone number of the caller (event value).

Figure 75 illustrates the actions performed by different entities when the telephony Widget is loaded to the user service environment. As illustrated, each *JS_CM* contains four lists: *L1*, *L2*, *L3*, and *L4*.

- *L1* is a list of the functionalities provided by the Widget associated to the *JS_CM* object. Each functionality is defined through a quintuplet *F (Goal, URL, Inputs data type, Outputs data type, HTML element)*.
- *L2* is a list of the data types generated by the Widget (including all functionalities) associated to the *JS_CM* object.
- *L3* is a list of functionalities provided by other Widgets (Widgets that are not associated to this *JS_CM* object), that match semantically (accept as input) a data type generated by the Widget associated to this *JS_CM* object.
- *L4* is a list of Widgets that generates data that match semantically one or several inputs of the Widget associated to the *JS_CM* object.

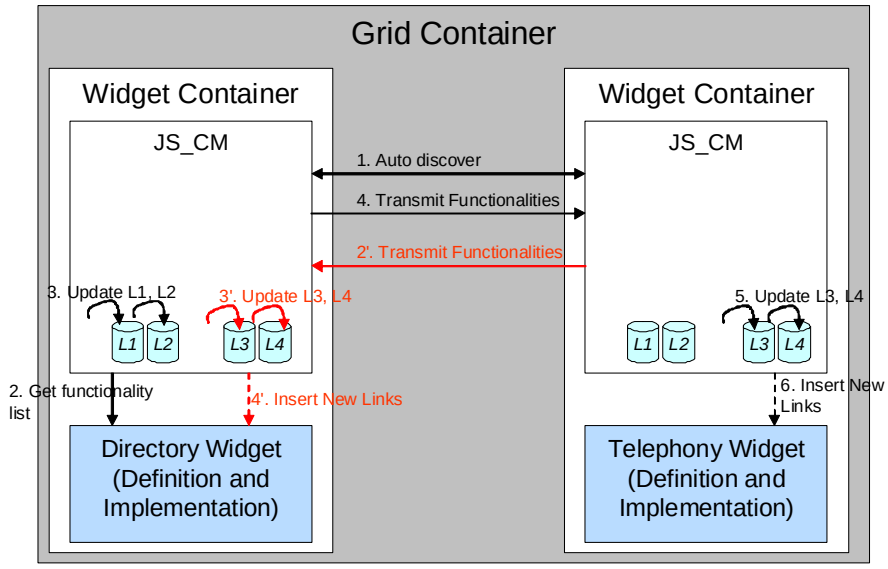


Figure 75: Widget initialization implementation.

When a Widget Container (WC) object is created, the embedded *JS_CM* object automatically discovers other *JS_CM* objects embedded in other WC objects (step 1). Following the discovery step, the *JS_CM* object corresponding to the new loaded Widget reads the Widget definition and gets its functionalities (step 2). Then, in step 3, it updates *L1* and *L2* (adding the functionalities description to *L1* and the list of generated outputs to *L2*). In step 4, it transmits the functionality list to all other *JS_CM* objects of other WC within the same environment. Each time a *JS_CM* object receives a list of functionalities provided by other Widgets, it updates its *L3* and *L4* lists if a semantic matching is detected (either between the data generated by the associated Widget and the inputs of the functionalities of other Widgets, or between the inputs of the functionalities provided by the associated Widget with the data generated by other Widgets) (step 5). If *L3* is updated with new entries (functionalities), the *JS_CM* object inserts an HTML element to the associated Widget (step 6); an HTML element which enables the user to launch a functionality of a Widget from another one (see the Widget running phase). This HTML element could an icon through which the user launches the destination functionality, or a drag and drop capability between the source Widget and the destination Widget (see Figure 77).

Figure 76 and Figure 77 illustrate the result of this phase; two Widgets linked through HTML elements. It illustrates a telephony Widget combined automatically with the directory Widget. Thus, when the user receives a call on the telephony Widget, an icon appears beside the phone number of the caller. This icon enables the user to automatically launch a search functionality on the directory Widget.

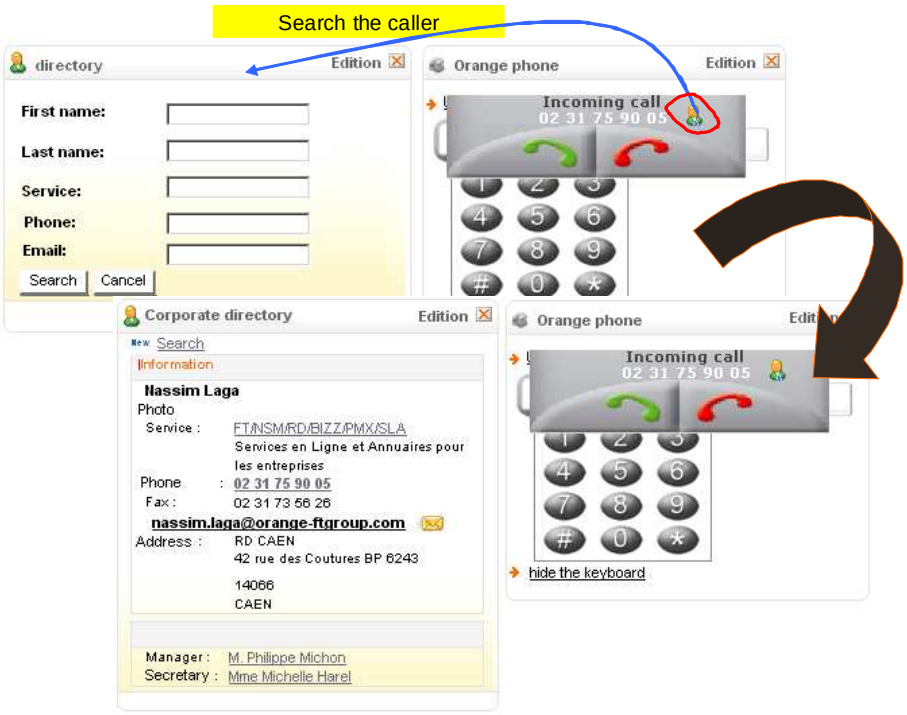


Figure 76: Automatic and semantic reuse of Widgets.

Figure 77 is another representation of links between Widget (the drag & drop capability).

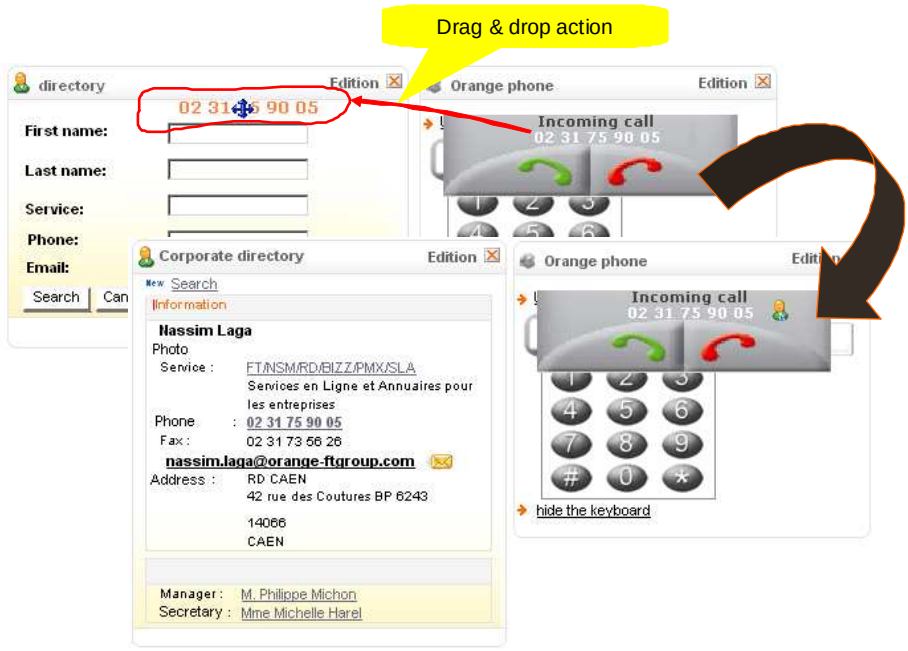


Figure 77: Link representation through a drag & drop capability.

b. Widget Running Phase

The Widget running phase corresponds to the step where the user can run the Widget (e.g. search a contact on the directory). During this phase, Widgets are automatically composed at the user initiative. In our example for instance (Figure 76), when the user receives a call on the telephony Widget, the framework proposes automatically to search the caller on the directory through an icon (HTML Element). When the user clicks on that icon, the directory search functionality is invoked, with the caller phone number as an input parameter. This sequence is illustrated in Figure 78.

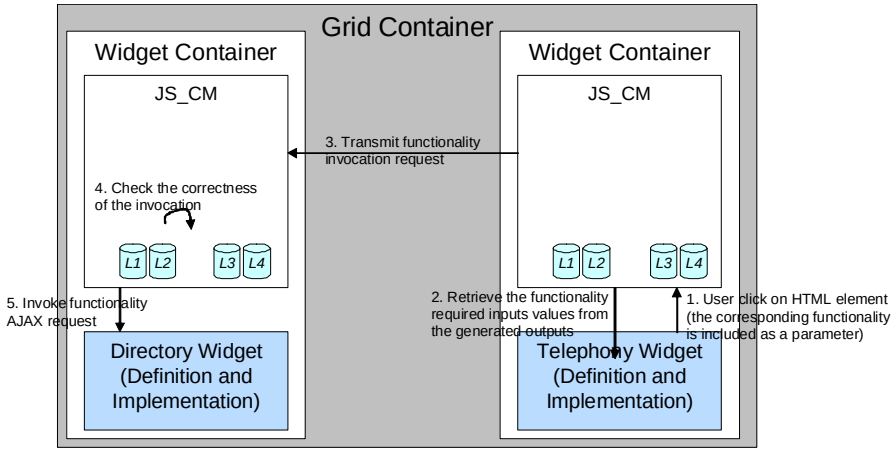


Figure 78: Widget communication implementation.

When the user clicks on the HTML element that has been inserted during the initialization phase, the associated *JS_CM* object is notified (step 1). Following this notification, the *JS_CM* object retrieves the data that are generated by the Widget and required by the invoked functionality (step 2). In step 3, the *JS_CM* object transmits to the *JS_CM* object of the destination Widget the functionality invocation request, which contains the functionality URL, and the required input data. The destination *JS_CM* object checks whether all required inputs are provided (step 4), and invokes the functionality using AJAX requests (step 5).

c. Widget Disconnection Phase

This phase starts when the user unloads a Widget from his environment. Figure 79 shows the different steps that are performed by the JS objects.

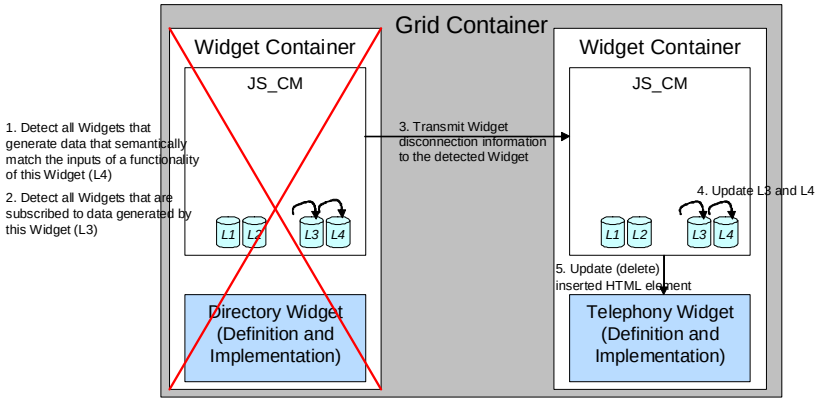


Figure 79: Widget Disconnection Phase.

First (step 1 and 2), the *JS_CM* object that corresponds to the unloaded Widget (Directory Widget in Figure 79) retrieves from *L3* and *L4*:

- the list of Widgets that provide functionalities those input match the data that are generated in the unloaded Widget,
- and the list of Widgets that generate data that semantically match the inputs of one or several functionalities provided by the unloaded Widget.

After detecting these two lists of Widgets, the *JS_CM* object transmits to the *JS_CM* objects of other Widgets the Widget disconnection information (step 3). Each *JS_CM* object that receives this information updates *L3* and *L4* (step 4), and optionally updates the UI of the Widget by deleting HTML elements that corresponds to functionalities of the unloaded Widget.

3.3 Process Manager Component

The goal of the Process Manager component is to enable the composition of Widgets based on a process (flowchart) definition. Thus, a Widget *A* can reuse capabilities of a Widget *B*, only if this combination is defined within the process. This component is implemented as JS object named *JS_PMC*. The current implementation is centralized, but it can be distributed as well. This component is instantiated with a process definition as an input parameter. In the following parts of this section, we will first detail the format that we use for defining a process; and second, we show how this process definition is executed. Though this process can be created manually, in the following Chapter we will illustrate the approach we introduce for the creation of this process; an approach which is intuitive enough to be used directly by users.

a. Process Definition

In the second part of this thesis we modelled a process through a graph *G (N, L)*, where nodes *N* represent the Widgets, and edges *L* represent links between them. In this section, we will show how we implement

this model. In our implementation, a process is defined using JSON format. Table 15 shows how we define nodes and links.

Table 15. Process Definition through a JSON format.

	JSON Format	Decsription
Nodes (widgets)	[{ widgetName: *value*, widgetIndexUrl: *value* }, …]	This is a JSON array. Each entry describes a Widget that is, or should be, loaded on the user environment. Ones a Widget is loaded, we also associate to the node a Widget instance id.
Edges (links)	[{ linkId: *value*, sourceWidgetName: *value*, sourceOutputType: *value*, destinationFunctionalityURL: *value*, destinationInputType: *value*, linkType: *value*, HTML_Element: *value* }, …]	This is a JSON array. Each entry describes a link between two Widgets. There are two types of links (automatic and semi-automatic), this is defined in *linkType* argument. The *sourceOutputType* enables the Process Manager to retrieve a specific output of a source Widget, and the *destinationInputType* enables to map the output to a specific input of the destination functionality (to be invoked through the *destinationFunctionalityURL*).

b. Process Execution

The Process Manager component (PMC) is implemented as a JS object (*JS_PMC*) integrated into the grid container object. This enables the control of all Widgets that are loaded on the same grid container. When the *JS_PMC* is instantiated with a process definition as an input parameter (step 1 on Figure 80), it first creates the Widgets (Widget containers) that are involved in the process definition; and second, it creates links between them according to the links defined within the process definition (steps 2 and 3). For each semi-automatic link, the *JS_PMC* creates an HTML element within the source Widget. For each automatic link, the PMC associates a listener to a Widget Container, which is in charge of checking the availability of the data those type matches the *sourceOutputType* used within the link. If such data is detected, the link will be executed automatically.

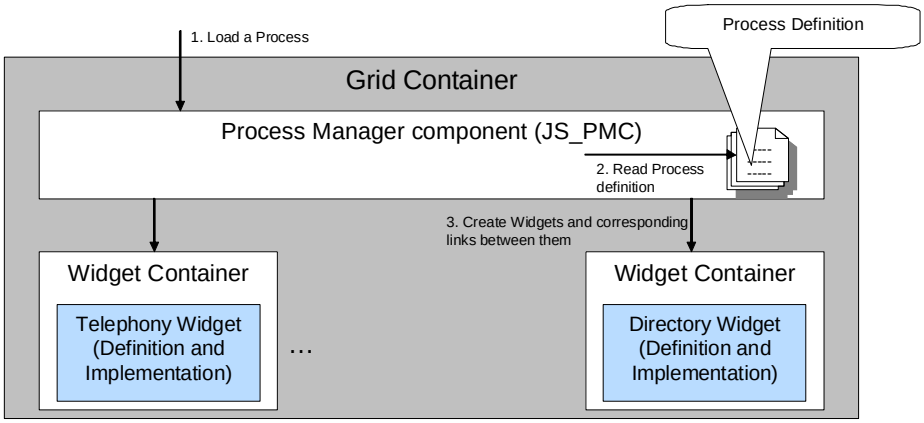


Figure 80: Process based linkage of Widgets

When a link (automatic and semi-automatic) is executed, a request is sent to the *JS_PMC* (step 1 in Figure 81). The *JS_PMC* checks that the link exists in the process definition, and gets the type of the data inputs that are required to invoke the destination functionality. Then, the *JS_PMC* retrieves the actual data values from the Widget (step 3), and invokes the destination functionality (step 4).

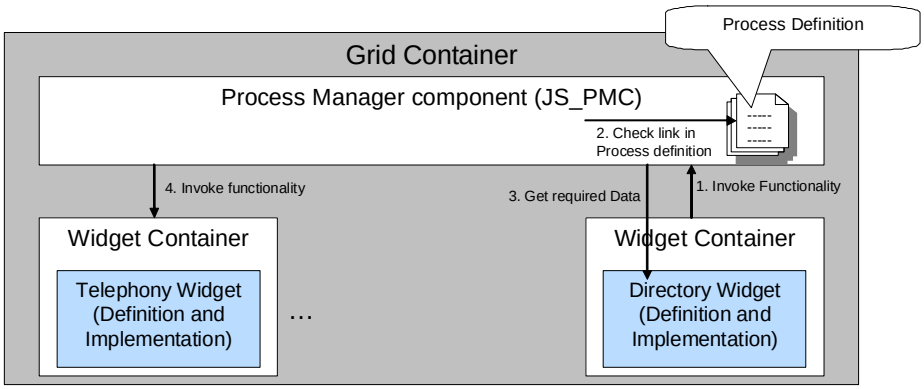


Figure 81: Link execution steps.

To illustrate the results, we show in Figure 82 two Widgets connected with each other through a process definition. The process includes two links. The first one is a semi-automatic link from the directory Widget to the telephony Widget. It enables calling a searched contact. The second link is an automatic link from the telephony Widget to the directory Widget. It enables to search a contact in the directory according to an incoming call phone number. Thus, each time a contact is displayed on the directory Widget, the framework proposes to launch the make call functionality of the telephony Widget; and each time there is an incoming call on the telephony Widget, the framework launches automatically the directory search functionality using the caller phone number as an input parameter.

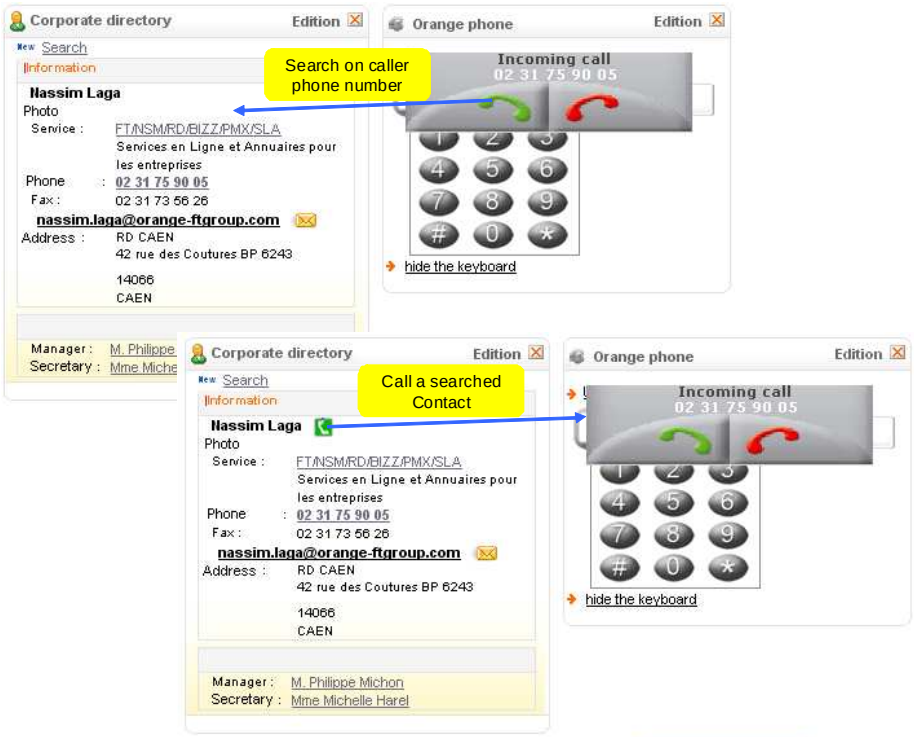


Figure 82: Illustration of a Process-based reuse of Widgets.

3.4 Abstract Service Based Reuse Extension

The abstract service based reuse of Widgets aims to first decouple service integrators and service providers; and second to provide a flexible runtime service selection based on criteria specified by the user himself.

The implementation of this mechanism is performed through two components: the Interpreter component and the abstract Widget. The Interpreter component is in charge of selecting the best Widget at runtime according to a needed functionality, and a set of selection rules. The abstract Widget provides a UI that enables the user to provide the inputs of the corresponding functionality, configure (choose) the selection rules to apply, invoke the selection process (Interpreter), and execute a selected Widget. The following subsections detail each component.

a. Interpreter

The interpreter component is implemented as Java Servlet. It is accessible through a URL using Get or Post method. Table 16 details the parameters that must be passed in the invocation. In addition to these parameters, the invoker may provide the available inputs (e.g. *destination_Phone_Number=0123456789*).

Table 16. Interpreter invocation details.

Parameter name	Value example	Description
user_id	*alice@host.com*	The identifier of the consumer of the abstract Widget. It enables the framework to get user specific data such as preferences and context.
format	*Json (or xml)*	This specifies the format of the output (list of selected Widgets) of the interpreter. Current supported formats are JSON and XML.
functionality	*Make_call*	The functionality of the abstract Widget.
constraint_rules	*[“$Context(identifier:knowledge.userId).location.country == selectedService.country;”, …]*	The list of constraint rules to apply during the selection process. The format is detailed below.
objective_rule	*MIN(selectedService.price);*	The objective rule to apply during the selection. The format is detailed below.

The interpreter component relies on a rule engine to evaluate constraint rules to true or false, and objective rule to a quantitative value. The rules may refer to static parameters such as price, and dynamic parameters such as location and presence status. Dynamic parameters are usually results of other services. As a consequence, it is important to enable referring to those services within the selection rules. The grammar of these rules we propose provides such functionality. Figure 83 is a simplified finite state machine diagram that defines the grammar. The grey states are legacy final states. This means for example a rule is considered as complete when we reach state S2 or S3.

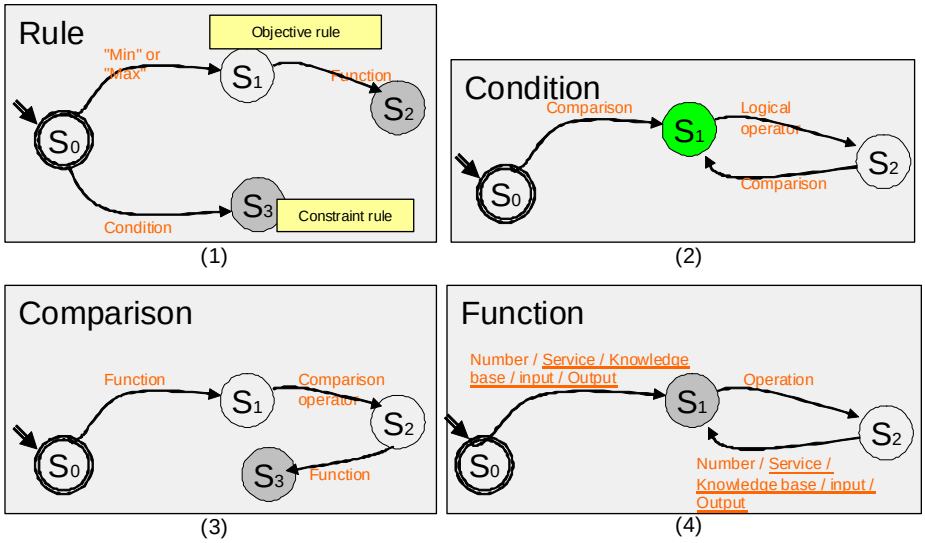


Figure 83: Rules grammar.

Figure 83.1 shows that a rule is either a constraint rule, or an objective rule. Objective rules contains necessarily an optimization operator (e.g. Max or Min), and followed by a function to optimize. A function (Figure 83.4) contains mainly numbers, variables, and operations. There are three types of variables:

- those referring to a knowledge base parameter such as price,
- those referring to the inputs provided by the user,
- and those referring to other services in order to get dynamic parameters values such as presence and location of the user.

Constraint rules (Figure 83.1) are simple conditions that are evaluated to true or false. Each condition (Figure 83.2) includes comparison statements connected with logical operators ('and' and 'or'). Each comparison statement (Figure 83.3) starts with a function (Figure 83.4), followed by a comparison operator ('<', '=', '>'…etc.), and ends with another function.

We have used LEX[35]/YACC[36] tools to generate a compiler and an evaluator for this grammar. More precisely, we have used JLex[37] and CUP[38] Java libraries.

b. Abstract Widget

The abstract Widget (Figure 84) is characterized by a UI that enables the user to enter the inputs, choose the rules to apply, invoke the Interpreter component to select the best available Widget according to the corresponding functionality and the selection rules chosen by the user, and invoke the selected Widget.

[35] Lex tutorial, http://dinosaur.compilertools.net/lex/index.html, accessed on June 11th, 2010
[36] Yacc tutorial, http://dinosaur.compilertools.net/yacc/index.html, accessed on June 11th, 2010
[37] JLex, www.cs.princeton.edu/~appel/modern/java/JLex/, accessed on June 11th, 2010
[38] CUP, www.cs.princeton.edu/~appel/modern/java/CUP/, accessed on June 11th, 2010

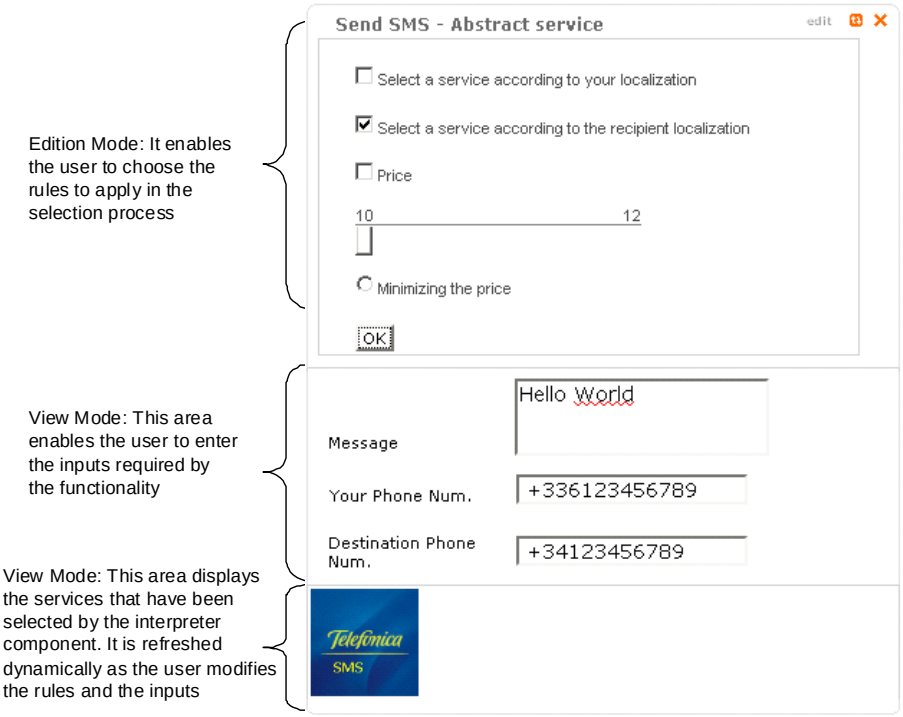


Figure 84: Illustration of a Send SMS abstract service Widget.

The abstract Widget is defined and implemented exactly in the same way we define and implement an ordinary Widget. It is defined using an XML format previously detailed (see Figure 68); and it is implemented using Web standards (XHTML, JavaScript, and CSS). The abstract Widgets are however provided by the Widget aggregator provider. It enables the Widget aggregator provider to wrap functionally equivalent Widgets available in the registry within a single abstract Widget. The actual Widget to be executed for a needed functionality is selected at runtime according to selection rules specified by the user himself. Figure 84 illustrates a *Send_SMS* abstract Widget. It shows the Edition mode, where the user can choose the rules to apply in the process of selecting the best Widget to execute. It shows the View mode, where the user can enter the inputs needed by the *Send_SMS* functionality (message, source phone number, and destination phone number). For each modification (in the values of the inputs or in the chosen selection rules), the Abstract Widget invokes the Interpreter component (Using AJAX request) to select the best available Widget according to the new parameters. The selection results (a JSON array generated by the Interpreter component) are displayed at the bottom of the Widget, where the user can click on a Widget to execute it and display the execution result.

3.5 Unstructured Data Based Reuse of Widgets

The unstructured data based reuse of Widgets aims to facilitate to service integrators (users and developers) the capturing of useful and unstructured data, and the reuse of such data in other services. In this section, we illustrate the implementation of such mechanism.

In our implementation we associate an unstructured data extraction module for each type of data that are likely to be generated by the Widgets. As a proof of concept, we used data extraction modules based on JS. This limits the extraction to the data that are accessible by the Web browser. In other words, multimedia content, usually seen by the browser as a black box, and accessible only through a heavy client, can not be considered (fetched) yet in this implementation.

Also, in our implementation, users or developers can associate a data extraction module to a Widget. This will enable the reuse of the unstructured data extracted from this Widget as input parameters in another Widget. For developers, they should just add a meta data in their Web page, in which they specify the type of data they want to extract (see Figure 85). For users, they can do this directly from their service environment at runtime (see Figure 86). In both cases, a listener is associated to the Widget UI, which is in charge of detecting the presence of the desired unstructured data and extracting them. This listener uses the JS module associated to the type of the data to be extracted.

The JS extraction modules must define *extract_Data* function. It receives as input a string (typically a Widget DOM inner HTML), and generates as output the same text in which the extracted data are tagged using their type tag (e.g. *tel*). When a JS extraction module is associated to a Widget, this function is called each time the DOM object of the Widget is modified.

```
...
<meta action="UnstructuredDataExtraction" id="EmailService" dataType="tel" />
<meta action="UnstructuredDataExtraction" id="EmailService" dataType="address" />
...
```

Figure 85: Adding of an unstructured data extraction module by a developer.

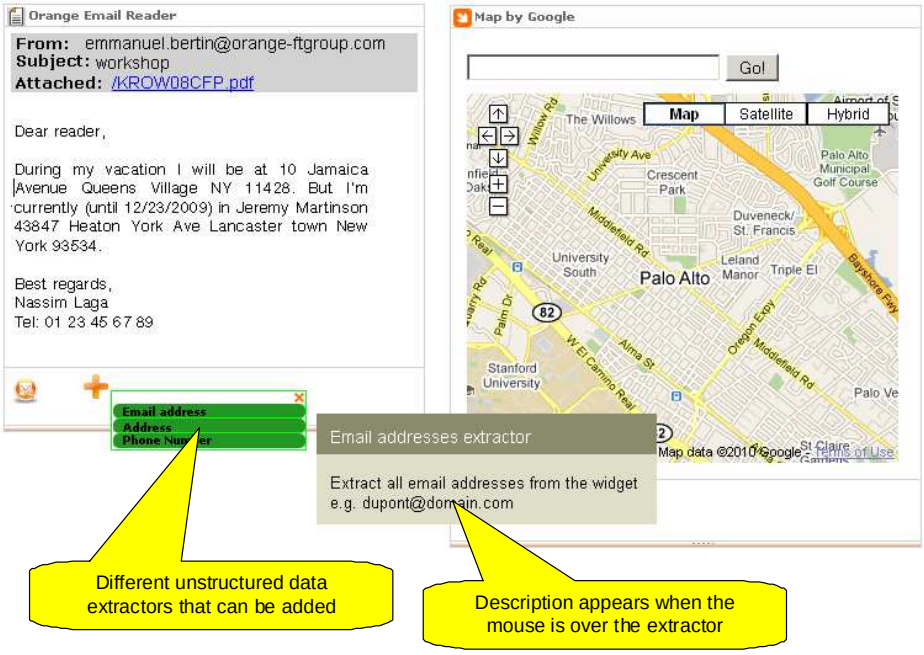


Figure 86: Adding of an unstructured data extraction module by a user.

Figure 87 illustrates how the extracted data could be reused as input parameters in other Widgets. The combination of the Widgets is performed either using the Automatic and semantic reuse mechanism, or using the Process based reuse mechanism (both are previously detailed).

Figure 87: Illustration of an unstructured data based reuse.

3.6 Cross-Device Reuse of Widgets

The cross-device based reuse of Widgets aims to extend the previously defined reuse mechanisms (API-based reuse, Automatic and semantic based reuse, and Process Based reuse) into a distributed environment, where different instances of the Widget aggregator run over different devices. In this section, we will detail the extension made to the Automatic and semantic based reuse mechanism.

This extension is characterized by the introduction of a new Widget, which is in charge of making the connection with other Widgets loaded on other devices. Let's refer to this Widget as a Cross Device Connecting Widget (CDCW). In the current implementation, the CDCW uses an authentication component to detect the devices of the same user. It also uses the cometD framework[39] (which is an implementation of the Bayeux AJAX push) in order to communicate between different CDCW loaded on different devices. The cometD framework is a publish/subscribe mechanism that enables to push asynchronous events to a client side (browser) of a Web application.

Basically, the CDCW capture the capabilities of Widgets that are loaded on the same device and publishes them to other devices; and similarly, it receives the capabilities of other Widgets loaded on different devices and publishes them into the Widgets loaded on the same device. Thus, CDCW uses the *JS_CM* (previously detailed) to interact with Widgets loaded on the same Widget aggregator instance; and it uses cometD framework to interact with other Widgets of other devices. Figure 88 shows a global view of the different components involved in this mechanism.

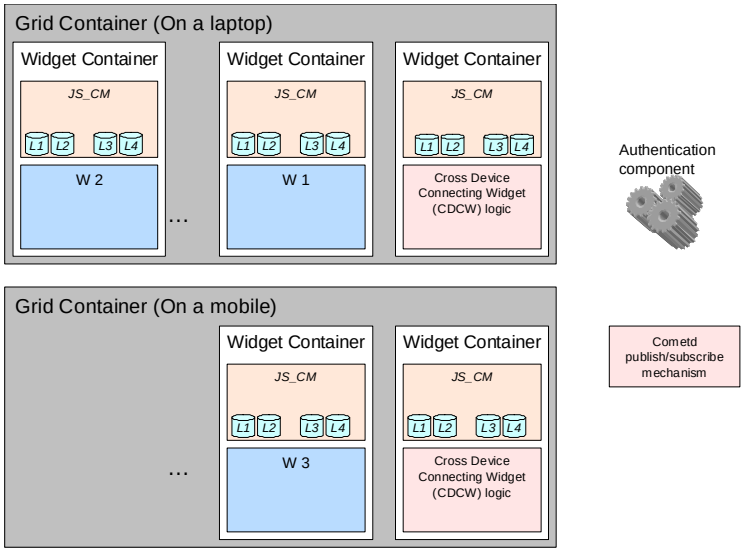


Figure 88: Component view of the cross device reuse mechanism.

Such as the automatic and semantic based reuse of Widgets, previously described, this mechanism manages different phases of the Widgets lifecycle. To describe this mechanism, we consider, as an initial state (Figure 88), that the user has:

- two instances of a Widget aggregator loaded on two different devices (e.g. a laptop and a mobile),
- already loaded two Widgets (W1 and W2) into his laptop instance,

[39] cometD, http://cometd.org/, accessed on June 14th, 2010

- already loaded one Widget (W3) into the mobile instance.

In the following parts of this section, we first illustrate the connection phase of (CDCW), and then we show the different states (connection, running, and disconnection) of an ordinary Widget.

a. CDCW Connection Phase

The CDCW plays the role of a bridge between the Widgets loaded on different devices. The first action performed by this Widget is to authenticate the user (Step 1 in Figure 89). Then, it creates, or joins, a communication channel created on the cometD publish/subscribe mechanism (Step 2). This channel is specific to one single user, and it enables the CDCW of devices belonging to the same user to exchange data. The channel identifier is in form of "*/userkey/**". The *userkey* is created after the authentication through a hash function on the user identifier (the result of the authentication procedure).

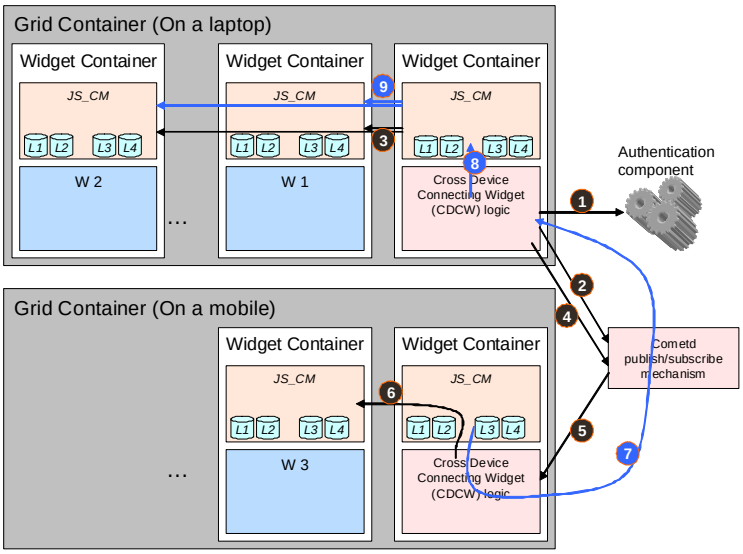


Figure 89: CDCW connection phase.

Once the communication channel is created on cometD, the *JS_CM* of the CDCW retrieves the capabilities of other Widgets loaded on the same device by exchanging with the corresponding *JS_CM* (Step 3). The capabilities of other Widgets are then transmitted to CDCWs of other Widgets using the previously joint cometD communication channel (Step 4 and 5). This transmission includes also an identifier of the device on which the Widgets are running. In step 6, the CDCW transmits the received capabilities to other Widgets loaded on the same device. This enables these Widgets to create links if a semantic matching is detected. Each link is defined as an octuplet *L (sourceWidget, outputDataType, destinationWidget, destinationDevice, Functionality, inputDataType, linkType, HTMLElement)*. This link is formatted using JSON as follows:

{
linkId: value,
sourceWidgetName: value,
sourceOutputType: value,
destinationDeviceId: value,
destinationFunctionalityURL: value,
destinationInputType: value,
linkType: value,
HTML_Element: value
}

As a response to the step 5, each CDCW sends back the capabilities of the Widgets that are loaded on the corresponding device (Step 7 and 8). Then, the Widgets that correspond to the CDCW which has just connected are notified by receiving the capabilities of the Widgets loaded on other devices (Step 9). This enables these Widgets to create links if a semantic matching is detected.

b. Ordinary Widget Connection Phase

When a user loads a new Widget (W4 on Figure 90) to his environment, the corresponding *JS_CM* reads the capabilities of this Widget and publishes them to other *JS_CM* of other Widgets of the same device, including CDCW (Step 1). The *JS_CM* objects that receive such information checks if is there a semantic matching, in which case they create a link. As a response to the step 1, the *JS_CM* of the CDCW sends back the capabilities of the Widgets that are loaded on other devices (step 2'). Then, The *JS_CM* of the CDCW transmits to other *JS_CM* of other devices the capabilities of the new loaded Widget (Step 2 and 3). Finally, the Widgets of other devices are notified by receiving this capability (Step 4), and if any semantic matching is detected a link is created between the two Widgets. The links are presented to the user through an HTML element as illustrated in Figure 91.

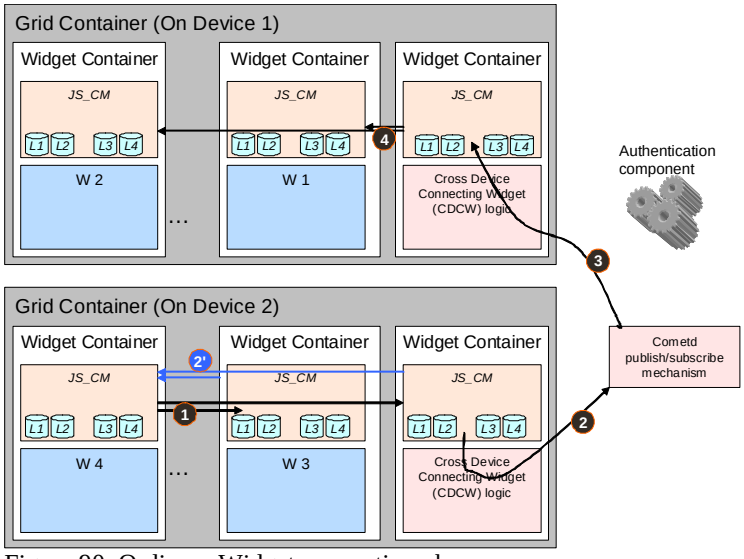


Figure 90: Ordinary Widget connection phase.

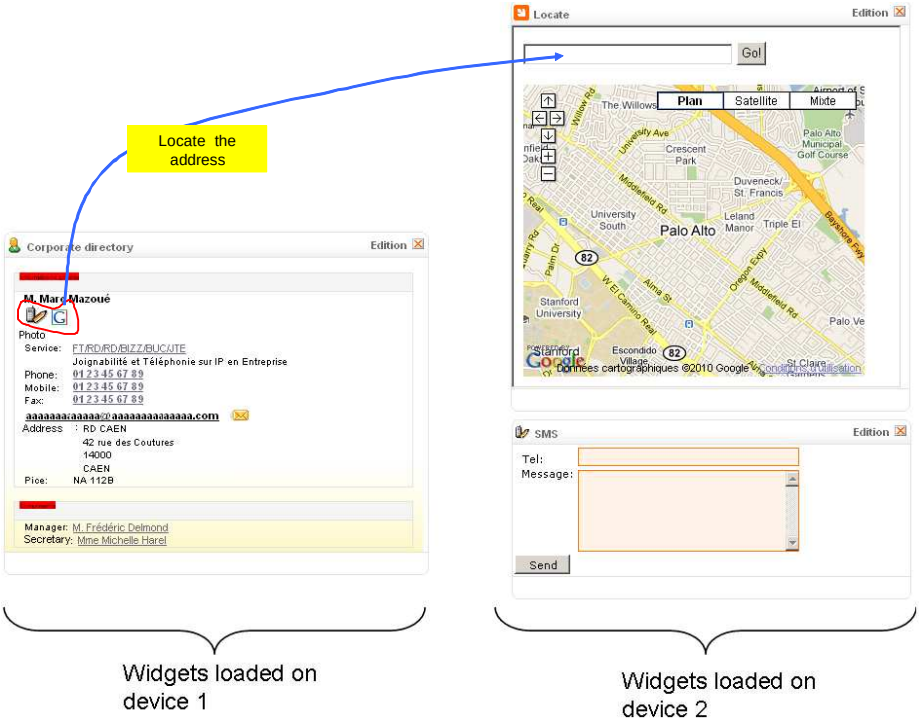


Figure 91: Cross device communication illustration.

c. Ordinary Widget Running Phase

When a user activates a link (by a click on the HTML element, created for example in W4 during the initialization phase), an event is sent to the *JS_CM* of the corresponding Widget (Step 1). The *JS_CM* is responsible of retrieving the data required for the execution of the link (Step 2) and their transmission to the *JS_CM* of the destination Widget. If the destination Widget is loaded on a different device, the transmission goes through the CDCW as illustrated in Figure 92.

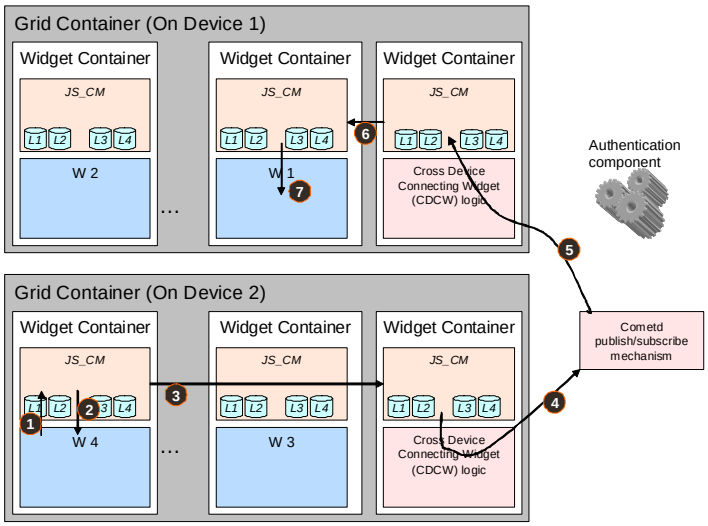


Figure 92: Communication process between two Widgets loaded on two different devices.

d. Ordinary Widget Disconnection Phase

As illustrated in Figure 93, when a Widget is removed from the user environment, the corresponding *JS_CM* transmits the information to other Widgets loaded on the same device, including the CDCW (Step 1). Then, each ordinary Widget updates it links. The CDCW however, transmits the information to other CDCWs loaded on other devices (Step 2 and 3). Each CDCW which receives such information broadcasts it over the Widgets loaded on the same device (Step 4). Finally, each Widget optionally updates the links (Step 5).

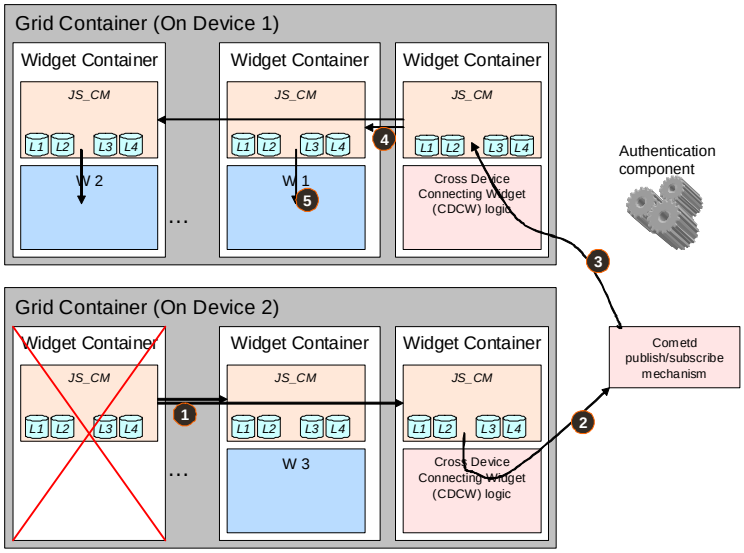


Figure 93: Illustration of the Widget disconnection phase.

3.7 Conclusions

In this Chapter we detailed the implementation of the Widget client and the different mechanisms it embeds. In summary, all defined reuse mechanisms are implemented at the front-end level using JavaScript language. The service selection mechanism we introduced (Interpreter) is however implemented as a Java servlet.

The lessons learnt at this stage are fourth.

- First, the microformat semantic dictionary is not sufficient to cover all data types that could be used in a given context. For instance, the microformat community did not standardized data types for emails (forwarder, recipient, subject, attachments, content, CC…etc), or calls (source phone number, destination phone number, start time, and end time). Therefore, it was necessary to define in some cases our own data types.
- Second, it is possible to integrate the reuse mechanisms we have defined in third party Widget aggregators, and even more, in ordinary web sites. Indeed, as the implementation of the reuse

mechanisms is distributed over the Widgets Container, the only condition needed is to embed the necessary JavaScript objects (those integrated in the Widget Container) to the implementation of the Widgets themselves. This could be performed by the Widget developer when creating the Widgets (at the design time), or dynamically by a proxy (at the Widget invocation time). A proof of concept is developed to show the feasibility of this mechanism; we have developed Widgets that first discover each other when they are present in the same Web page (e.g. iGoogle or Netvibes), and second, they enable the user to combine them.

- Third, the Widget combination based on a process definition remains basic compared to existing flowchart based composition mechanisms we have detailed in Chapter I.1 State of the Art. Indeed, in the process based Widget combination, it is not possible to define conditions and loops within the process definition. This is rather a choice than a limitation. Indeed, it is important to simplify as much as possible the creation of the process definition to be understandable by ordinary users.
- Fourth, the current implementation does not enable the different Widget combination mechanisms to retrieve data from two different Widgets, and invoke a third one with that data. For example, it does not enable retrieving a phone number from a contact list, and a string from an email, and invoking a send SMS Widget with the phone number and the string as input parameters. To realize this use case, first the user should execute two links (one from the contact list Widget to the send SMS Widget, and one from the email Widget to the send SMS Widget); and second, the send SMS Widget should save the data received in the execution of the first link and use them during the execution of the second link.

Chapter III.2 Illustration of WOA in Different SOA Application Fields

Following the detailed implementation of the WOA, this Chapter aims to illustrate its usage within the two application fields we consider: service composition and business process management. In addition to illustrating how WOA is applied to these fields, we also compare it with SOA.

1 Service Composition

Before illustrating the different approaches of service composition using WOA, we first sketch a scenario and then detail how it is realized using the different approaches of service composition: static service composition, semi-automatic service composition, and automatic service composition.

1.1 Driving Scenario

The scenario we rely on for demonstrating the advantages and the limitations of the WOA in service composition field is characterized by three actors within a company: Charlie the CEO of the company, Alice the secretary of Charlie, and Bob a team manager.

Bob wants to discuss with Charlie about a new project. First, he needs a directory Widget in order to search the contact information of Charlie's secretary, Alice. Second, he needs to find the best service (Telephony, IM, email…etc) to contact Alice. The best service depends essentially on Bob's own criteria; though these criteria may take into account Alice's preferences and context. A typical criterion for selecting the best service could be formulated as follows: select the service according to the presence status of the recipient, and if several services are candidate then select the cheapest one. For the rest of the scenario, let us suppose that the IM service has been selected (manually or automatically) to communicate with Alice.

From Alice point of view, she needs to anticipate some actions each time she receives a communication invitation (e.g. receive an IM, incoming calls…etc). Such actions could be the display of the contact information of the initiator of the invitation, or the display of previous exchanges of emails with that person.

Bob wants to schedule a physical meeting with Charlie. Therefore, several date proposals would be exchanged between Bob and Alice in order to agree on a specific slot. Each time a new proposal is sent and/or received, the recipient is likely to check the agenda availability. After being agreed upon a date,

Alice initiates an agenda invitation (proposed by the service environment) on behalf of Charlie. Charlie and Bob accept the meeting.

Half an hour before meeting Bob, Charlie is in another important meeting, and he is afraid that it will last longer than expected. Therefore, he wants to contact Bob in order to apologize and propose him to shift the meeting to the lunch time at a restaurant. He first retrieves the contact information of Bob, either from the meeting participant list (Agenda), or from the directory service. Second, Charlie selects (manually or automatically) the communication service to use. Consider that the IM service has been selected, as Bob is at his office and present on the IM using his laptop. After apologizing and agreeing on shifting the meeting, and having it at the restaurant, Charlie loads the public directory service (e.g. yellow page) to his laptop in order to search a restaurant. Charlie sends a selected restaurant address to Bob. Charlie also displays the itinerary to go to the restaurant on a Map loaded on his mobile device. When Bob receives the address, he likely locates it using a Map service, and loads the underground Map service in order to know how to go there. Ideally, the Map service and the underground Map service would be displayed on the mobile phone, as Bob will move afterward.

As illustrated in Table 17, this scenario presents several manual compositions of services. In the next subsections, we will see the different options provided by the WOA to automate them.

Table 17. Manual composition list of the scenario.

Composition	Actor	Transited data	Goal
Directory and IM	Bob	Email address (legacy output of Directory service)	Send an IM
Directory and Telephony	Bob	Phone number (legacy output of Directory service)	Make call
Directory and Email	Bob	Email address (legacy output of Directory service)	Send an Email
Directory and Send SMS	Bob	Phone number (legacy output of Directory service)	Send an SMS
IM and Directory	Alice	Email address (legacy output of IM service)	Search IM initiator
Telephony and Directory	Alice	Phone number (legacy output of Telephony service)	Search Call initiator
IM and email Inbox	Alice	Email address (legacy output of Telephony service)	Display Email exchange history with IM initiator

IM and Agenda	Alice	Date (not a legacy output of the IM service)	Check Charlie's availability
IM and Agenda	Bob	Date (not a legacy output of the IM service)	Check his availability
Directory and IM	Charlie	Email address (legacy output of Directory service)	Send an IM
Agenda and IM	Charlie	Email address (legacy output of Directory service)	Send an IM
Public directory and IM	Charlie	Postal address (legacy output of the public directory service)	Send the restaurant address
Public directory (on laptop) and Map (on mobile)	Charlie	Postal address (legacy output of the public directory service)	Display the itinerary to the restaurant address on a map
IM and Map	Bob	Postal address (not legacy output of the IM service)	Locate the address received in the IM
IM and Underground Map	Bob	Postal address (not legacy output of the IM service)	Retrieve how to go to the address received in the IM

1.2 Static Composition

The main features highlighted in the contribution part of this thesis concerning the static service composition are the personalization capability provided to the users and the loose coupling using the concept of abstract Widget. To illustrate these two features let us consider the composition of the directory service with respectively the IM service, the Telephony service, the send email service, and the send SMS service (the four first lines of Table 17).

a. Personalization

There are two ways to create a customizable composite service using the WOA. The first method is characterized by constructing a Widget which invokes other Widgets when present within the same environment. The second method is characterized by relying on the concept of abstract Widgets.

In our example, in the first method developers create a directory Widget in which they first check the presence of respectively the IM Widget, the telephony Widget, and the send email Widget in the user environment; and second compose it with them. Figure 94 shows a code snippet of a directory Widget that first discovers Widgets that are present in the user service environment, checks the presence of a telephony

Widget and a send email Widget, inserts icons if that Widgets are present, and invokes the destination Widget when the user clicks on an inserted icon.

```
var WidgetList = Widgetcombination.API.getWidgetList();
for(var i = 0; i < WidgetList.length; i++){
  if(WidgetList[i].goal = "Make_call"){
    var div = document.getElementById("actions");
    div.innerHTML = div.innerHTML + "<img src='http://.../makeCall.ico' "+
                                    "onclick=\"invokeWidgetFunctionality('"+i+"')\"/>";
  }else{
    if(WidgetList[i].name = "Send_email"){
      var div = document.getElementById("actions");
      div.innerHTML = div.innerHTML + "<img src='http://.../sendEmail.ico' "+
                                      "onclick=\"invokeWidgetFunctionality('"+i+"')\"/>";
    }else{
      if(WidgetList[i].name = "Send_SMS"){
        var div = document.getElementById("actions");
        div.innerHTML = div.innerHTML + "<img src='http://.../sendSMS.ico' "+
                                        "onclick=\"invokeWidgetFunctionality('"+i+"')\"/>";
      }
    }
  }
}
function invokeWidgetFunctionality(index){
  Widgetcombination.API.publish(WidgetList[index].WidgetId, WidgetList[index].functionality, "tel",
                                document.getElementById("tel").innerHTML);
}
...
```

Figure 94: Code snippet of the directory Widget.

This approach for composing Widgets is customizable because the user still has the control on the Widgets that could be loaded to his service environment. As a consequence, the directory Widget will not be composed with the telephony Widget if the user does not load them into the same service environment. Figure 95 shows how does the user personalize the composite service realized in Figure 94.

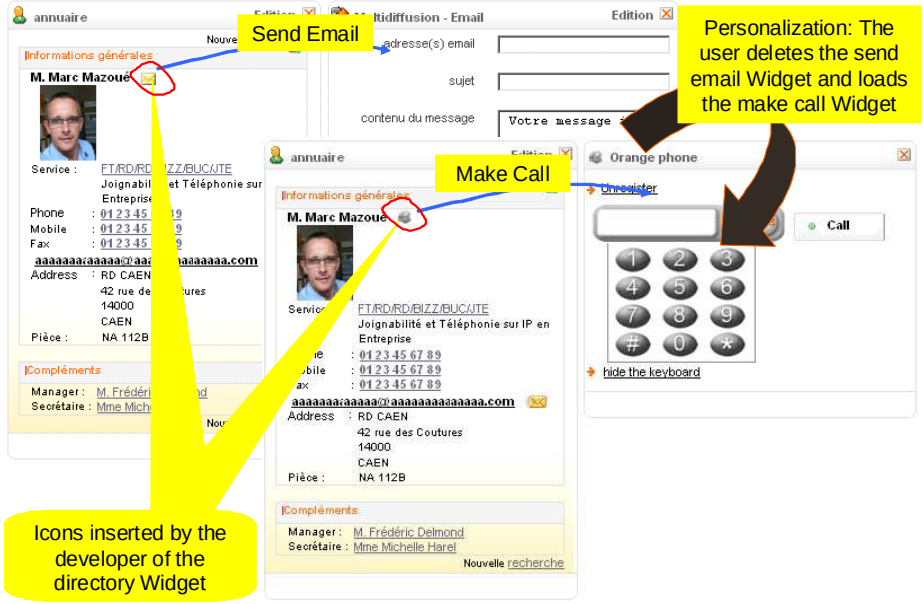


Figure 95: Manual Personalization.

The second method for creating a customizable composite service is to rely on the concept of the abstract Widget. Instead of creating a directory Widget which is connected to the different communication Widgets (e.g. send SMS, send IM, send Email), the developers links the directory to the different needed functionalities, namely send SMS, send IM, and send Email. The service environment is then in charge of selecting the best Widget available at the runtime. This method is customizable because the user can influence the selection process by configuring himself the criteria that define the best Widget.

From the developer point of view, the code will be exactly the same as the previous one illustrated in Figure 94. And from the user point of view, they should first load abstract Widgets into their service environment; and second, configure the selection rules to apply. Figure 96 illustrates the directory service composed with the send SMS abstract Widget.

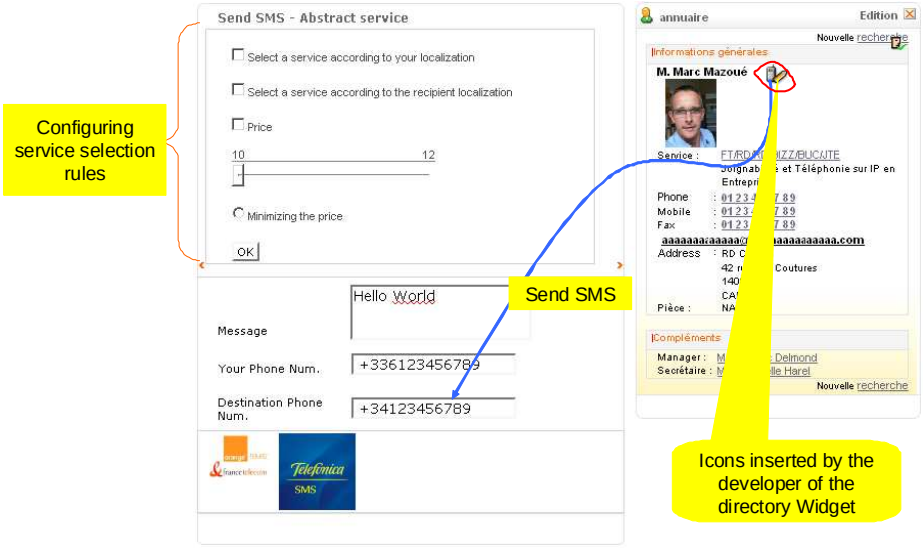


Figure 96: Abstract Widget based personalization.

The two personalization approaches are similar from the developer point of view. However, from the user perspectives, the abstract Widget based personalization is automatic while the other is manual (deleting the Widget and replacing it with another). But, the abstract Widget alone does not enable the user to personalize the logic of the composition. In our example for instance, the send SMS abstract Widget will never select a make call Widget to contact a person displayed on the directory. Therefore, the two personalization approaches complement each others. Thus, the manual personalization enables the selection of the functionalities to be composed with the directory Widget, and the abstract Widget selection mechanism selects the best Widget (among functionally equivalent ones) to be invoked according to the user's own criteria.

b. Loose coupling

WOA enables developers to create composite services independently of the used basic services. They do this by relying on the abstract Widget concept exactly in the same way we illustrated in Figure 96. In that Figure, the directory Widget is completely independent from the send SMS services it uses. It is linked to the send SMS abstract Widget, but not directly to the different send SMS Widgets such as "Orange Send SMS" or "Telefonica send SMS". In other words, if Orange withdraws the send SMS Widget from the repository, the composition is still valid as the framework will automatically select, at runtime, another Widget fulfilling the same functionality. The only condition is the availability of at least one send SMS Widget within the registry that satisfies the selection rules defined by the user.

1.3 Semi-automatic Composition

The key features introduced by the WOA in the semi-automatic service composition field are: enabling ordinary users to compose services, the cross-device composition by users, the unstructured data based composition, and the loose coupling between composite services with the used basic services. To illustrate these features, we rely on the compositions detected in our scenario (Table 17).

a. User service composition

As we specified in *Chapter II.3 Widget-Oriented Architecture (WOA) in SOA application fields*, the WOA facilitates significantly the process of creating a composite service. This is performed through two steps. The first one is the creation of links based on semantic matching, and the second one is characterized by enabling the user to personalize the created links (delete them, or modify their type). In order to illustrate this feature, let us consider Alice needs in the previous scenario. Let us focus on her needs of connecting the IM Widget with the directory Widget, the telephony Widget with the directory Widget, and the IM Widget with the email Widget.

From the developer point of view, the only requirement is developing the services as Widgets, providing a functional and non-functional description, and annotating the outputs at the UI level. Figure 97 for instance shows the description file of the directory Widget. It shows a description of two functionalities: searching by email address, and searching by phone numbers.

```
<?xml version="1.0" encoding="windows-1250"?>
<module name="directoryFT" indexUrl="http://../services/directory/">
  ...
  <functionality  name="searchContact"
                  actionUrl="http://../services/directory/dir.php"
                  label="search FTRD"
                  goal="readDirectory"
                  iconUrl="http://../img/person.ico"
                  overviewURL="http://../directory.jpg">
    <inputs>
      <data name="PhoneNumber"      type="tel" />
    </inputs>
    <outputs>
      <data name="contactInformation"   type="hCard" />
    </outputs>
  </functionality>

  <functionality  name="searchContact"
                  actionUrl="http://../services/directory/dir.php"
                  label="search FTRD"
                  goal="readDirectory"
                  iconUrl="http://../img/person.ico"
                  overviewURL="http://../directory.jpg">
    <inputs>
      <data name="emailAddress"     type="email" />
    </inputs>
    <outputs>
      <data name="contactInformation"   type="hCard" />
    </outputs>
  </functionality>
  ...

</module>
```

Functionality declaration

Functionality declaration

Figure 97: Directory description file snippet.

Figure 98 shows the microformats annotations used to annotate the outputs of the directory Widget.

```
<?xml version="1.0" encoding="utf-8"?>

<div class="vcard">
  <span class="fn">
    <span class="given-name">Nassim</span>
    <span class="family-name">Laga</span>
  </span>

  <div class="org">FT/RD/RD/BIZZ/BUC/JTE;Joignabilité et Téléphonie sur IP en Entreprise</div>

  <span class="tel">
   <span class="type">work</span>
   <span class="value">02 31 75 90 05</span>
  </span>

  <span class="tel">
   <span class="type">fax</span>
   <span class="value">02 31 73 56 26</span>
  </span>

  <a class="email" href="mailto:nassim.laga@orange-ftgroup.com">nassim.laga@orange-ftgroup.com</a>

  <span class="adr">
    <span class="street-address">42 rue des Coutures</span>
    <span class="postal-code">14000</span>
    <span class="locality">Caen</span>
    <span class="country-name">fr</span>
  </span>
</div>
```

First and last name

Organization

Phone number

Fax number

Email address

Postal address

Figure 98: Microformats annotations.

From the user point of view, it is necessary to load the needed Widgets into the same environment (the same tab in the Widget aggregator we defined). The Widget combination component we detailed

creates automatically links between the Widgets according to their description file. This is illustrated in Figure 99.

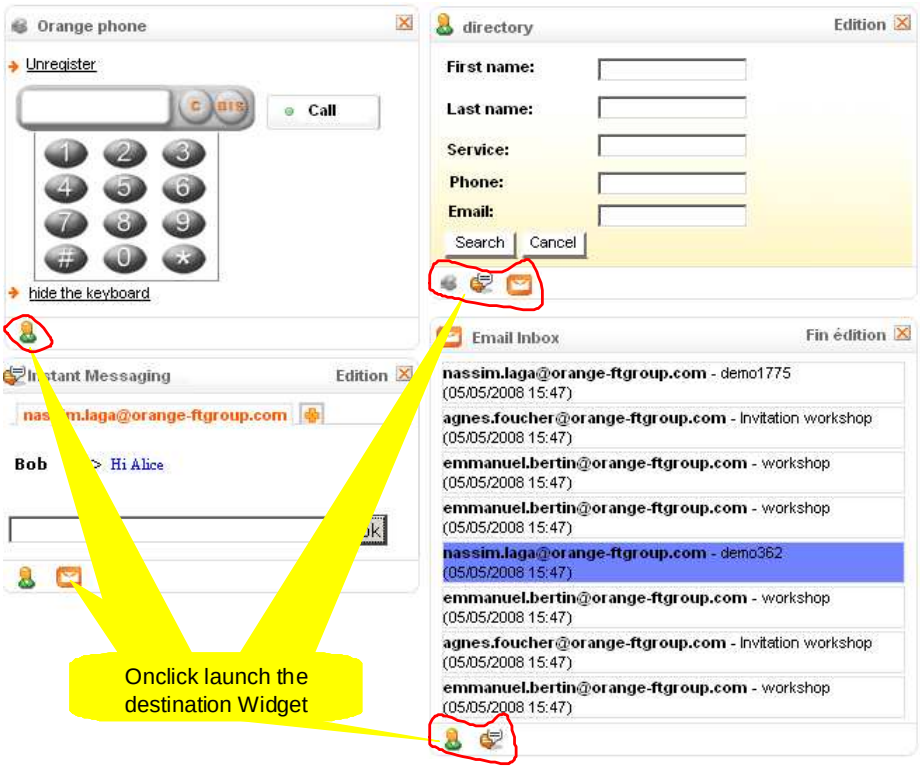


Figure 99: Semantic matching based linkage of Widgets.

Figure 99 illustrates also two limitations. The first one is that some links, which are not needed by Alice, have been created (e.g. the link between the email inbox Widget to the send IM Widget is not needed by Alice (select an email and initiate an IM session with the forwarder)). The second limitation is that the created links are not automatic. Alice must click on the link to launch the destination Widget, though in some cases this could be automated (e.g. for each incoming call the directory search on the caller could be launched automatically).

The second step of user service composition we proposed using WOA aims to tackle these two limitations. For each link, the framework adds two additional GUI elements that enable the user to delete or automate the corresponding link. Figure 100 illustrates this feature. It shows a link between the telephony Widget to the directory Widget, which can be modified using two icons: one for deleting the link, and another for automating the link.

From the technical point of view, the first step creates a composite service ($G<N,L>$) which connects all connectable services (based only on semantic matching), and the second step enables the user

to refine the composite service definition by deleting undesired links, and automating the frequently used links.

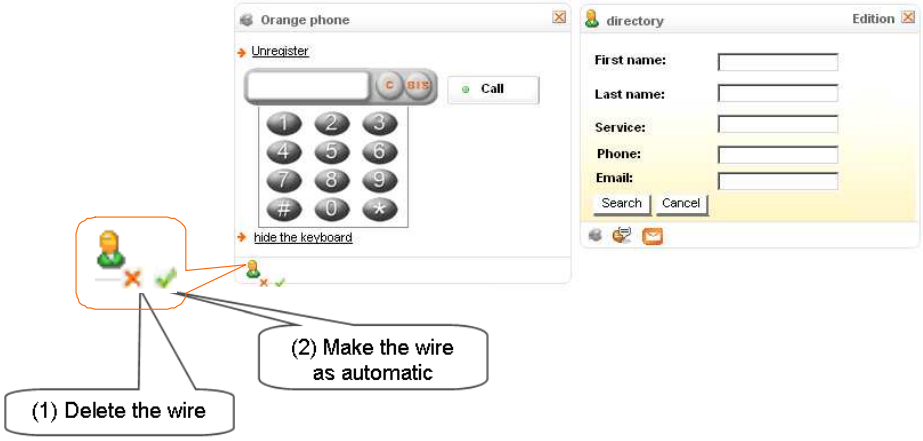


Figure 100: Composite service personalization.

b. Cross-device composition

To illustrate this feature, let us consider the manual composition performed by Charlie when he invokes the Map service loaded on the mobile with the restaurant address generated by the public directory service loaded on his laptop. The WOA provides to the user the capability of automating this composition. It enables the user to define a composite service that combines Widgets loaded on different devices.

From the developer point of, there are no additional actions to perform to enable such composition by the user. However, the users must first load the *Cross Device Connecting Widget* (CDCW) on all devices they want to connect. This Widget authenticates the user, and detects and connects all instances of the Widget aggregator loaded on different user devices (illustrated in Figure 101).

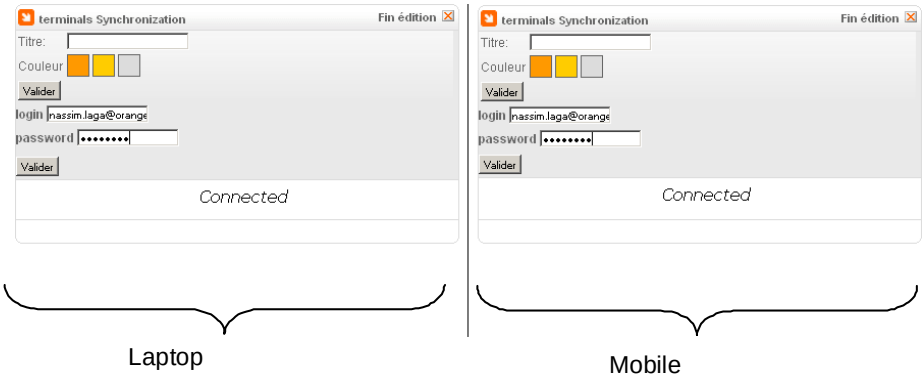


Figure 101 : Authentication Widget.

Second, the user loads the different Widgets he needs on the different devices. As illustrated in Figure 102, the framework connects automatically the Widgets that are semantically compatible (based on

their description). Third, such as the user service composition mechanism detailed in the previous subsection, the framework enables the user to modify or delete links created during the second step.

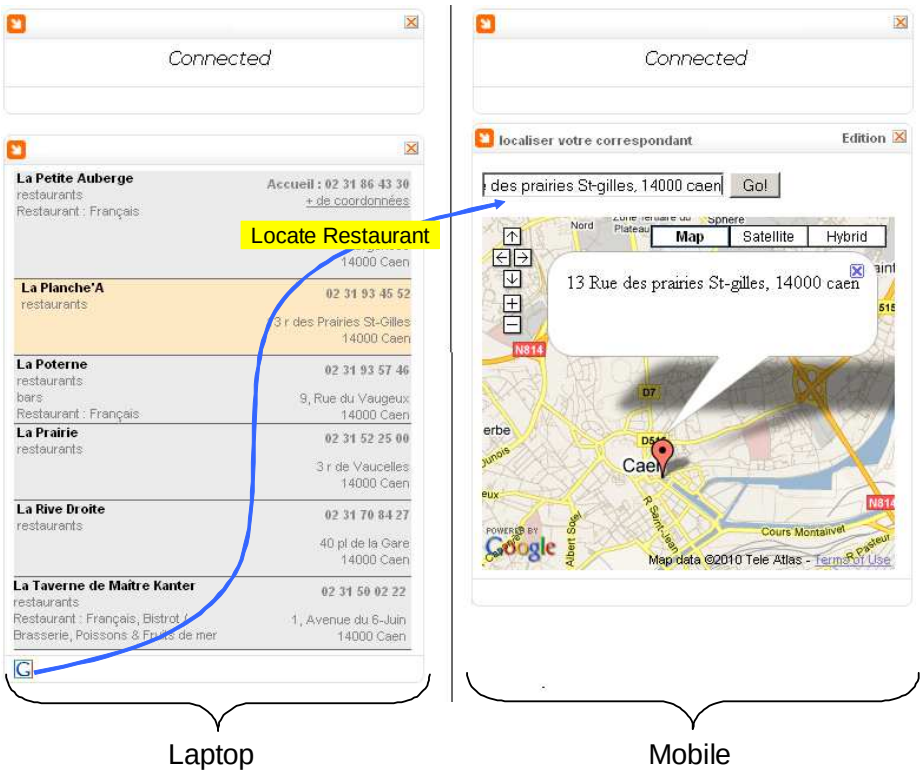


Figure 102: Cross device composite service.

c. Unstructured data composition

To illustrate this feature, let us consider the manual composition made by Alice or Bob when they have composed the IM service with the agenda service; by extracting the exchanged dates and checking their availability on the agenda. Current SOA based composition tools do not enable ordinary users to make such composition, as they rely on connecting only legacy inputs and outputs of services. The WOA provides the necessary infrastructure to enable users to easily combine services (Widgets) based on unstructured data generated in different services (e.g. communication services).

The process for making such composition does not require additional actions from the developer. The only need is the description of the different Widgets. From the user point of view, the actions to perform can be summarized in two steps. First, the user loads the two Widgets into the same environment (or in two different by connected instances of the environment). Second, through a GUI, the user associates a data extraction module for the source Widget. To extract dates for instance, the user associates the date extraction module as illustrated in Figure 103. After associating the date extraction module, the framework automatically detects and extracts dates within the Widget. For each date detected, the framework detects

automatically the Widgets whose input is compatible with that data; in our case, it detects the agenda Widget that enables the user to check his availability at a given date passed as input parameter. If a semantic matching is detected between two Widgets, the framework creates a link that enables the user to combine the Widgets. When, the user clicks on the link, the destination Widget (agenda) is launched automatically.

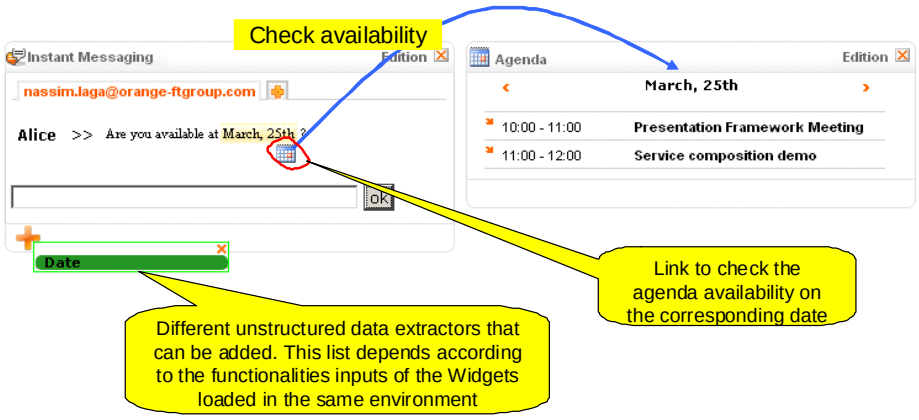


Figure 103: Unstructured data based composition.

d. Loose coupling

As we detailed in *Part II Contributions*, the WOA enables a loose coupling between composite services and the basic services they use. This loose coupling is realized either manually as users now can create their own composite service, or automatically through the concept of abstract Widget.

The manual approach is characterized by the replacement (by the user) of a Widget by another that offers the same functionality. The framework then first detects semantic matching between the new Widget and the others that are loaded in the same environment, and second creates links accordingly. These two actions reconstitute the previous composite service.

The automatic approach is characterized by relying on the concept of abstract Widget; which has been illustrated in *1.2 Static Composition*.

1.4 Automatic Composition

The main feature introduced by WOA concerning the automatic service composition is the failure recovery system. This is pertinent as current composition tools based on natural language still lack of accuracy due to the limitations of natural language processing tools. To illustrate this feature, let us consider the manual composition performed by Charlie when he searches a restaurant (composition of the public directory service with the Map service). This composite service can be created through a natural language request (e.g. search and locate restaurant), using WOA as well as SOA. However, the WOA provides the capability

of modifying the created composite service to match exactly the requestor needs. In our case for example, Charlie would likely modify the created composite service so that the Map Widget will run on his mobile instead of the laptop, as he will need the Map as he goes to the restaurant. To do that, he has just to delete the map widget from his laptop, and load it on the mobile; the framework will make the connection automatically based on semantic matching. This process is illustrated in Figure 104.

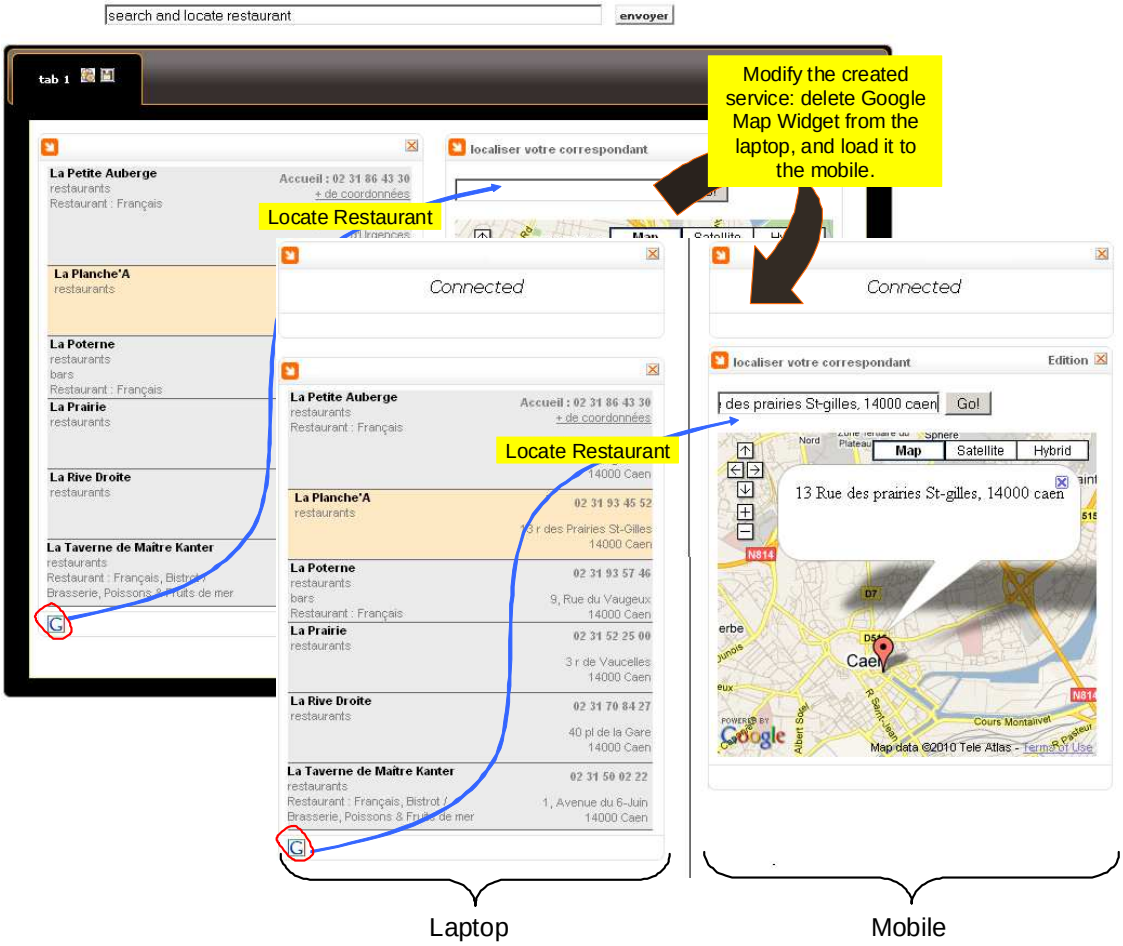


Figure 104: Failure recovery process in automatic service composition using WOA.

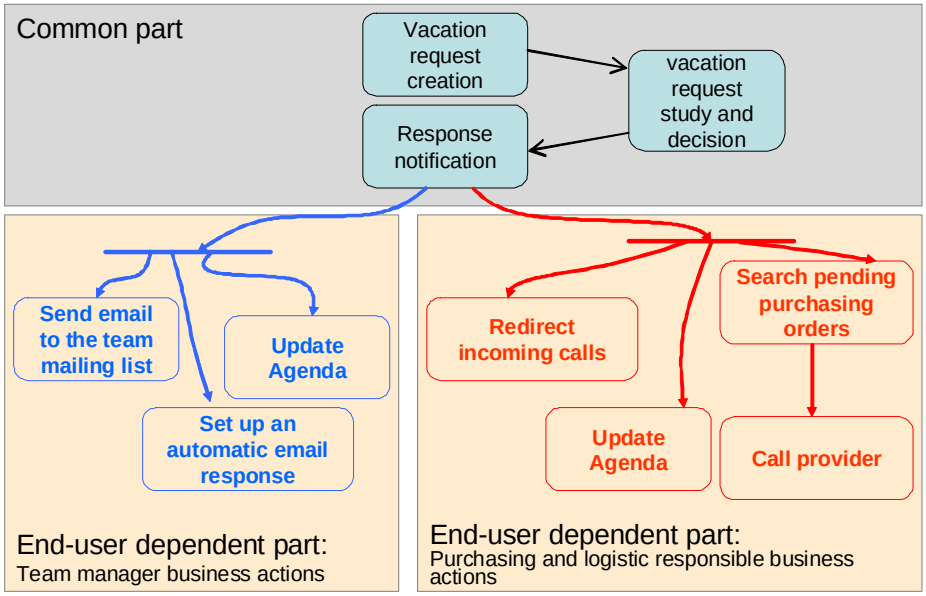
2 Business Process Management (BPM)

The key features introduced by WOA to BPM are: a solution for business processes heterogeneity and dynamicity, the capture of unstructured data, and the loose coupling between process definitions and the used concrete services. As business processes are usually implemented as composite services, the capture of unstructured data and the loose coupling are previously detailed in section *1.3 Semi-automatic Composition*. Therefore, the goal of this section is to illustrate only the solution brought by WOA to tackle

the heterogeneity and dynamicity of business processes. For this purpose, we rely on a driving scenario detailed in the next subsection.

2.1 Driving Scenario

As an illustrative scenario we consider the vacation request business process of two employees. It is previously illustrated in Figure 62 which is duplicated hereafter. This Figure depicts two versions of the vacation business process; one performed by a team manager, and another performed by a purchasing and logistic responsible. This business process has a part which is common and mandatory to all users, and another part which is user dependent. The former consists of the vacation request creation, the vacation request study and decision (by the requestor manager), and the response notification. The latter consists of updating agenda, sending email to the team, and setting up an automatic email response for the team manager; and searching pending purchasing orders, calling product providers, redirecting incoming calls during the vacation period, and updating agenda for the purchasing and logistic responsible.



The user dependent part of business processes is heterogeneous and dynamic at the same time. It is heterogeneous as it differ from a user and another, and it is dynamic as the company needs to adapt quickly for new businesses from one hand, and the user himself could change his habits from another hand. In the next subsection, we will see how this heterogeneity and dynamicity is tackled using the WOA.

2.2 Heterogeneity of Business Processes

The main feature introduced by the WOA is the service composition capability provided to ordinary users. Relying on this capability, we proposed a method for managing the heterogeneity of business processes. This method is characterized by splitting business processes into a common part and a user dependent part.

- Business analysts are then in charge of modelling the common part (which usually contains mandatory activities needed for performing a given business goal).
- Developers are in charge of automating the common part as a Widget.
- Finally, users are in charge of defining and automating (using semi-automatic composition) their own part (user dependent part).

Figure 105 illustrates how the vacation request business process is automated (including the common part and the user dependent part) for the purchasing and logistic responsible.

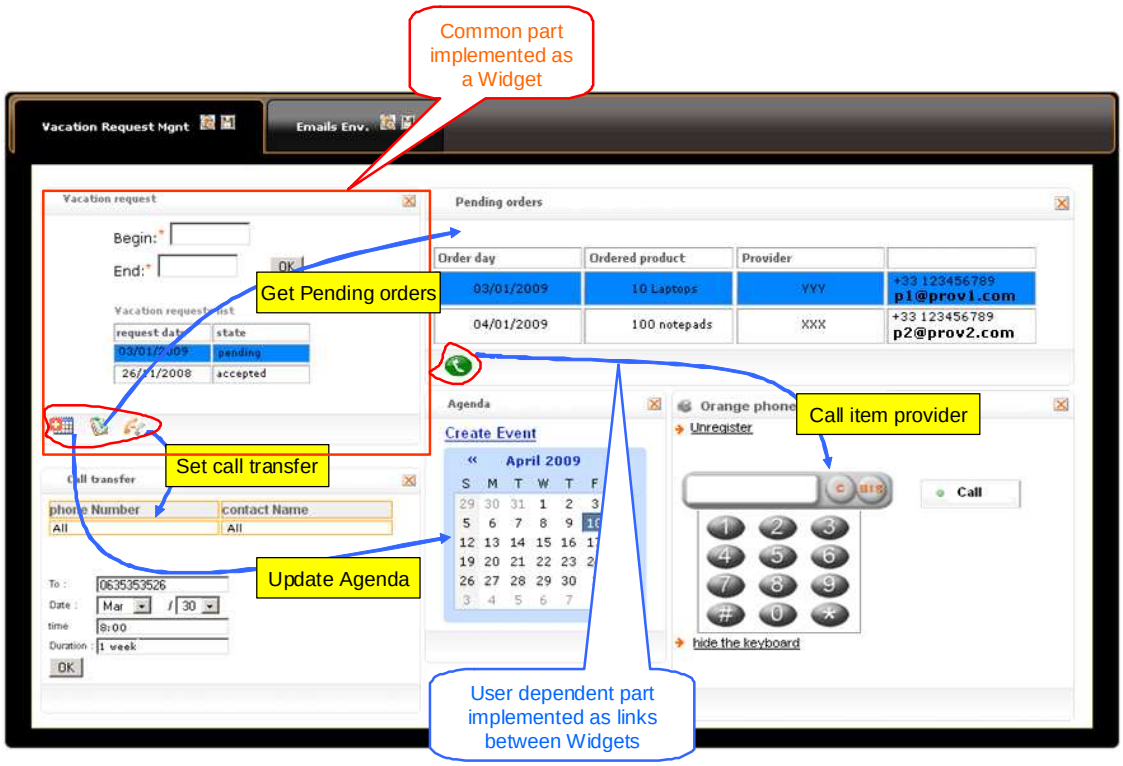


Figure 105: Business process implementation example.

The use of the WOA enables the scalability of the process of modelling and implementing business processes from one hand; and it enables a high automation of the business processes from another hand. Indeed, business process management in WOA is scalable because business analysts (resp. developer) do not care about the details of the processes, which are specific to users; instead, they model (resp. implement) only parts of business processes which are common to all users. The WOA enables a

high automation of business processes as the user dependent part, which can hardly be captured and implemented by business analysts and developers, is automated by the users themselves.

In addition to these two features, the WOA provides a good infrastructure for business analysts to capture user habits, and optionally modify the common part if common actions have been detected.

2.3 Adaptation of Business Processes

Unlike the common part of business processes, which is persistent, the user dependent part is usually very dynamic. In order to manage such dynamicity, organizations can either rely on the semi-automatic composition capability provided to users, or use the concept of abstract Widget. Both methods have been previously detailed and illustrated. In summary, semi-automatic composition of services enables users to define themselves the user dependent part of a business process. More important, it enables them to modify it as they face new businesses, contexts, and needs in general. Though the modification is manual (adding new activities by adding new Widgets, chaining the activities (Widgets) between them, and deleting obsolete activities), it is performed at runtime and enables the modification of the logic of the business process. In our example for instance, if the Purchasing and logistic responsible wants to set up an automatic email response during his vacations, he has just to load the corresponding Widget, and personalize the links with the other Widgets. This is illustrated in Figure 106.

The second approach for adapting a business process to new needs is to rely on the abstract Widget concept. In other words, users will load abstract Widgets instead of ordinary Widgets. By doing so, the platform will automatically select at the runtime the best available Widget, for performing a given activity in the business process. Let us consider for instance the step where the purchasing and logistic responsible call the provider of a given item. The change in the communication service (telephony service) used by the company would oblige the user to manually change the business process implementation. However, using the concept of abstract Widget, the platform will automatically adapt the business process and take into account the new telephony service.

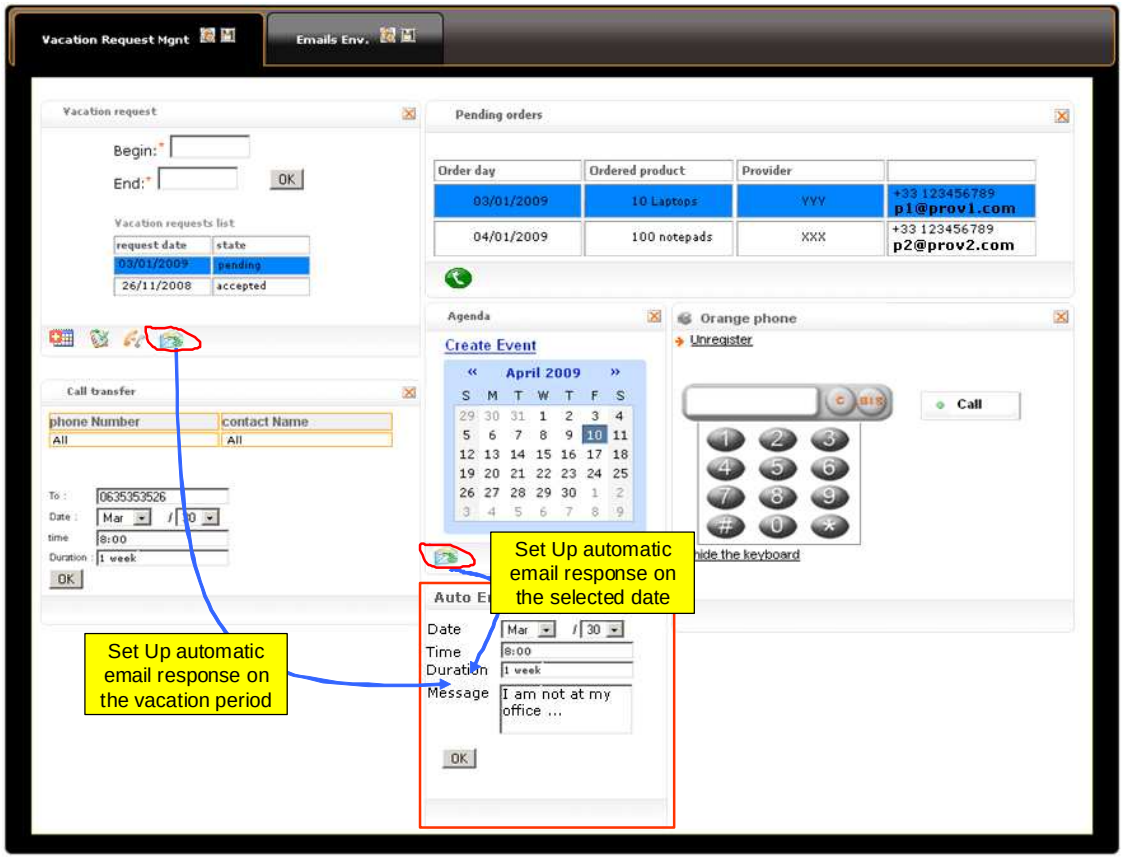


Figure 106: Adaptation to a new need.

3 Conclusion

In this Chapter, we illustrated how the WOA paradigm and the Widget aggregator we designed and implemented can be used by developers and users in service composition and business process management fields. Table 18 summarizes the different challenges we addressed, the involved roles in the solution we propose, and what should be performed by each role. Basically, in service composition, the developer is in charge of creating the different Widgets and providing a description file for each one; and the user is in charge of discovering the Widgets and loading them into his service environment (Widget aggregator). In static service composition, the developer is also in charge of discovering the Widgets that are loaded at runtime by the user, and using their capabilities.

In BPM, we have proposed to split business processes into a common part and a user specific part. Associated to the WOA, this enables to tackle the heterogeneity and the dynamicity of business processes. The cost of this approach is a new task for users and another one for developers. Developers should automate the common part of a business process through a Widget; and users should model and automate the user specific part of the business process.

Table 18. WOA impacts on service composition and BPM involved roles.

Field	WOA advantages	Developer tasks	User tasks
Static Service Composition	- Enabling users to personalize a composite service. - Loose coupling (when using abstract Widgets).	- Creates services as Widgets. - Provides a description file for each Widget. - Within the Widget code, he uses the Widget Combination API to discover Widgets loaded to the aggregator and to use their capabilities.	- Discovers and loads Widgets to the Widget aggregator. - Ideally, users load abstract Widgets to the Widget aggregator.
Semi-automatic Service Composition	- Enabling users to create composite services (Intuitiveness of the composition tool). - Loose coupling (when using abstract Widgets). - Cross device composition.	- Creates services as Widgets. - Provides a description file for each Widget.	- Discovers and loads Widgets to the Widget aggregator (optionally on different instances and different devices). - Personalize the links (created automatically) between Widgets. - Ideally, users load abstract Widgets.
Automatic Service Composition	- Existence of a Failure recovery system when a created composite service does not match exactly the user needs. - Loose coupling.	- Creates services as Widgets. - Provides a description file for each Widget.	- Makes a natural language request. - Refine a created composite service by adding new Widgets, removing others, and optionally modifying the links that have been created.
Business process	- Automation of heterogeneous	- Implement the common part of business processes	- Loads the Widget that implements the common

Management	processes. - Rapid adaptation to new needs. - Unstructured data capture. - Loose coupling	(modelled by business analysts) as a Widget. - Creates other services related to business activities of users as Widgets. - Provides a description file for each Widget.	part of a business process. - Discovers and loads other Widgets that are needed to perform the user specific part of the business process. - Optionally, personalizes the links that have been created automatically.

Chapter III.3 Experimentation and Dissemination

The Widget aggregator we have defined and implemented was exposed to users. In this Chapter, we summarize the experimentations/demonstrations contexts and the feedback collected on the different mechanisms.

1 Experimentation by Orange Labs Staff

In order to get the users' feedback about the advantages and limitations of the Widget aggregator and the Widget Combination mechanisms, we have deployed our implementation within the Orange Labs for 184 participants. Among this population, 66% are familiar with Web 2.0 portals, and 33% discover this type of applications for the first time. In the experimentation, we have developed some functionalities of applications as Widgets; applications that are used inside the company. A comprehensive list of those widgets is presented in Table 19. It is important to note that in this experimentation the links that are created between the Widgets by the Widget Combination component are represented to the user through a drag & drop capability between the Widgets (The drag & drop mechanism was illustrated in Figure 77).

Table 19. List of Widgets tested by users.

Widget name	Widget functionality description	Original application
Professional_Email_Inbox	Checking and reading emails from the professional email service	Microsoft Exchange
Send_Email	Sending emails	Microsoft Exchange
Contact_List	Checking the contact list	Microsoft Exchange
Check_Agenda	Checking the events inside the agenda	Microsoft Exchange
Add_To_Agenda	Add a new event to the agenda	Microsoft Exchange
Communicator_Presence	See the presence status of contacts	Microsoft Communicator
Communicator_IM	Instant messaging	Microsoft Communicator
Conference_Bridge	Creating and searching a conference call bridge	Internal application for managing conference calls
Web_Conference_Pilot	Managing the conference bridge (Check the connected phone numbers, put someone to mute…)	Internal application for managing conference calls

Web_Phone	Telephony	Orange telephony REST API
Send_SMS	Send SMS	Orange send SMS REST API
Send_MMS	Send MMS	Orange send MMS REST API
Search_Directory	Searching someone in the company directory	Internal directory
Book_ Meeting _Room	Book a meeting room for a specific slot of time	Internal application that manages the meeting rooms
Gmail_Inbox	Checking and reading emails of Gmail	Gmail
Map	Locate addresses	Google Map
GTalk	Instant messaging	Google Talk
Google_Search	Use the Google search engine	Google search
Translator	Text translation	Google translate
Public_Directory_Search	Search in a public directory	118712 directory (http://www.118712.fr/)
File_Storage	A remote file storage system	A remote file storage system
Video_Search	Searching videos inside the company intranet	Company search engine
Video_player	Playing videos	Video player

After five months of experimentation, the results show clearly the usefulness of the Widget aggregator in the enterprise context. Indeed, 89% have used the Widget aggregator platform, 80% have created their account and personalised their environment by loading only the needed Widgets, and 55% have used, or intend to use, the aggregator framework as a default starting page of their Web browser. In addition, 72% consider the aggregation platform useful and 14% consider it necessary.

The Widget aggregator we experimented embeds a Widget Combination capability. The users feedback concerning this capability was also positive and encouraging. Indeed, as illustrated in Figure 107, 54% of the participants have used the Widget Combination capability. Users have even proposed to implement the functionality between other widgets than those that already support it.

However, there were 17% of people that did not discover the existence of the Widget combination mechanism, and there were 9% who were aware about it but do not use it. The explanations provided by users could be summarized in two conclusions: first, the drag & drop representation of links is not very intuitive as users that are not aware about its availability between two Widgets can hardly discover it; second, it is necessary to implement much more Widgets (related to users business activities) to really benefit from the Widget Combination capability.

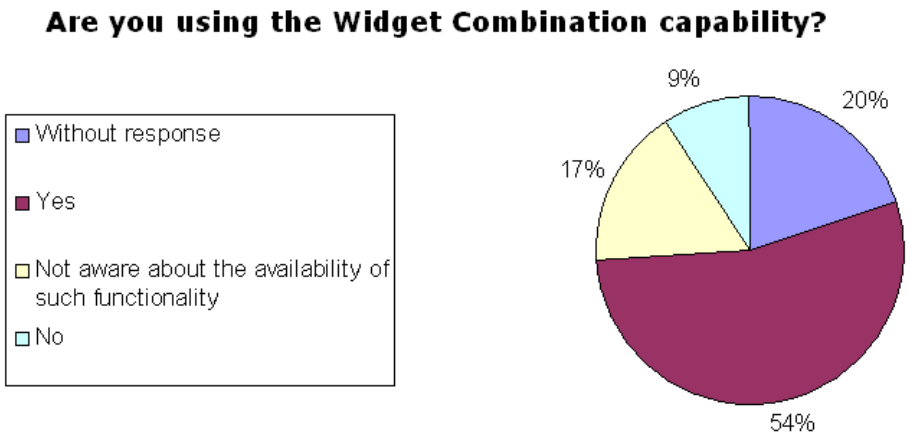


Figure 107: Users feedback about the Widget Combination capability.

2 Demonstration to Marketing Team of Orange

The Widget Combination capability has been demonstrated to a marketing team; responsible of the Orange Widget aggregator product (available at http://www.espace-utilisateur.orange-business.com/). Based on Figure 108, two scenarios have been shown. In the first scenario, the user has a meeting entry on his agenda. The Widget aggregator automatically propose to the user to locate the meeting address (using *Google Map* Widget), to display the contact information of the organizer (using the *Directory* Widget), and finally to book a meeting room (using the *Book_Meeting_Room* Widget). Let's suppose that the user locates the meeting address and notices that he will be late. Therefore, he decides to contact the organizer and propose him/her to have a conference call, instead of a physical meeting. To do that, the user needs first to get the contact information of the organizer (using the *Directory* Widget); second, contact him/her using for example a *send_SMS* Widget; third, book a room for the conference call (using *Book_Meeting_Room* Widget). The Widget aggregator has automatically linked these Widgets with each to facilitate these actions to the user. Thus, by a simple click, the user displays the contact information of the organizer of the meeting. Then, through a simple click the user composes the *Directory* Widget with the *send_SMS* Widget to contact the organizer. Finally, the user can also book a meeting room on the meeting time slot by composing the *Agenda* Widget with the *Book_Meeting_Room* Widget.

In the second scenario we have demonstrated, the user receives a call on the telephony Widget, and the Directory Widget is automatically launched to search for the caller information.

The feedback collected during this demonstration is very positive as the marketing team decided to include the Widget Combination capability in the next version of the Orange Widget Aggregator. In

addition, the same demonstration was proposed to and accepted at the Orange Labs research exhibition in December, 2008. However, two important remarks have been raised:

- The first remark is about the automatic chaining of the telephony Widget and the directory Widget. The attendees have reported that this automation could be intrusive for the user. Therefore, it is important to enable him to deactivate this automation.
- The second remark is about the chaining of several Widgets (Agenda → Directory → Send SMS), which is a kind of a process that should be performed by the user to achieve a given goal. Therefore, it could be interesting to link this mechanism to business processes of the user to perform more automation, without being intrusive.

These remarks motivated the process-based reuse of Widgets mechanism we have defined latter on.

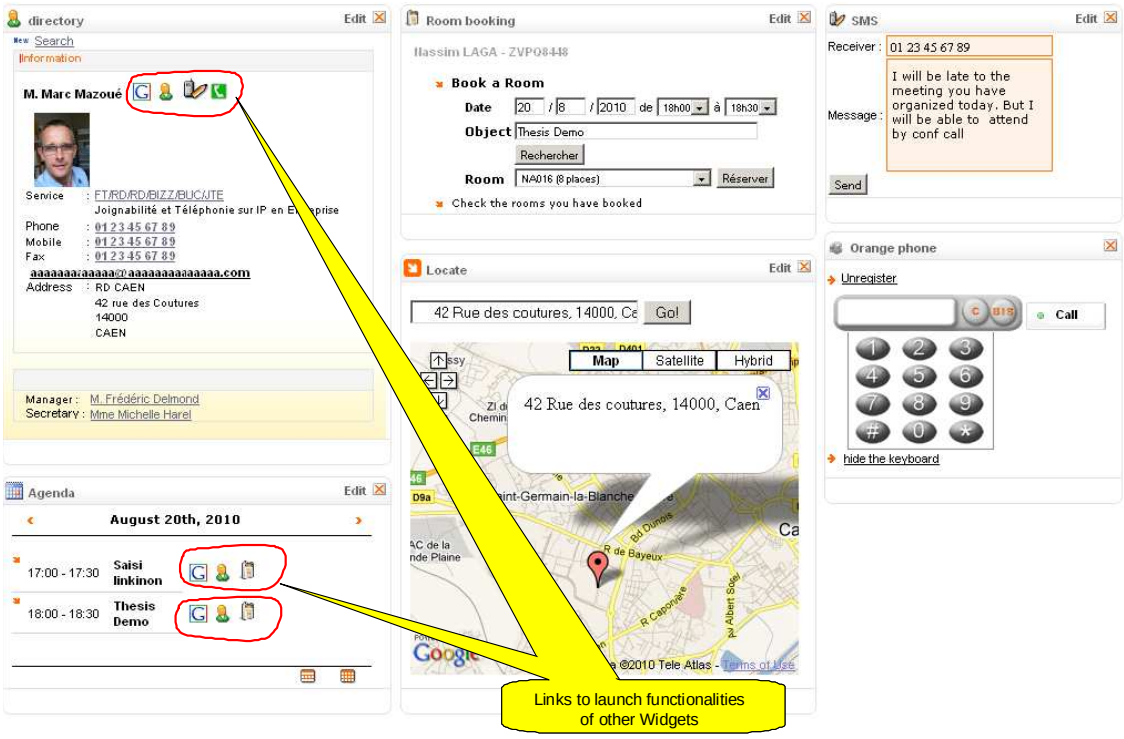


Figure 108: Screenshot of the demonstration to the marketing team.

3 Integration within SERVERY Project Contributors

The concept of the abstract Widget has been demonstrated within the European project SERVERY (Service Platform for Innovative Communication Environment). The project aims to build a marketplace of converged services (Telecom and Web services), where service creation, service management, and their execution on heterogeneous platforms is supported. The demonstration shows several abstract Widgets.

Each abstract Widget wraps two or three real services. Table 20 shows a comprehensive list of abstract Widgets that are implemented, the real services which are considered, and the selection rules that could be applied in each abstract Widget. Figure 109 is one screenshot of the demonstration.

Table 20. Abstract Widgets list.

Abstract Widgets		Considered Services (Service that are wrapped within the abstract Widget)
Functionality	Rules	
SendSMS	• Selection according to recipient location. • Selection according to sender (user) location. • Selection according to the delivery guaranty level. • Selection according to the price per SMS.	• Orange Partner Send SMS API (http://www.orangepartner.com). • Telefonica Send SMS API (http://open.movilforum.com)
GetCalendar	- Selection according to the activity context of the user (work/home).	- Google Calendar. - Microsoft Exchange Calendar (accessible only within Orange private network)
GetEmails	- Selection according to the activity context of the user (work/home).	- Gmail. - Microsoft Exchange emails (accessible only within Orange private network).
GetContacts	- Selection according to the activity context of the user (work/home).	- Gmail contacts. - Microsoft Exchange contacts (accessible only within Orange private network).
GetWeather	• Selection according to user preferences. • Selection according to user location. • Selection according to the service price.	• Meteo France • WeatherForcast (http://www.webservicex.net)
SearchPictures	• Selection according to user preferences. • Selection according to the user language. • Selection according to the service price.	• Microsoft bing search engine. • Flickr search. • Picasa search.
Translate	• Selection according to user preferences.	• Google translate. • Microsoft bing translator. • Yahoo babel fish translator.

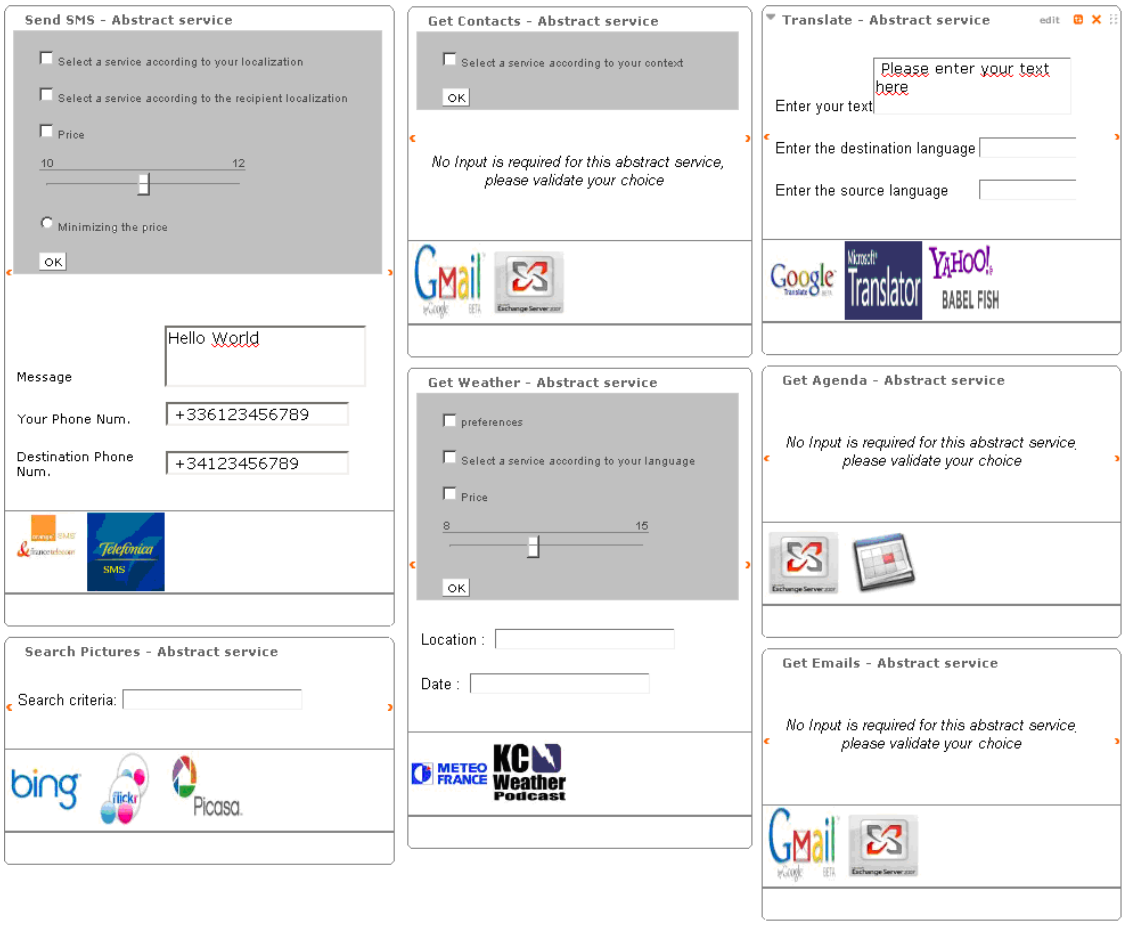


Figure 109: SERVERY demonstration.

The feedback collected is positive as first the concept was included as part of the demonstrated mechanisms of the midterm review of project. Second, the concept was integrated to different service composition components defined by the project, namely natural language composer, and Open Mashups Studio[40]. The goal is to first decouple composite services from the basic services they use; and second perform runtime adaptation of composite services according to rules that can refer to heterogeneous parameters such as user context, preferences, Qos…etc. Third, the concept was proposed an accepted for demonstration at the Orange Labs research exhibition (June 2010).

This demonstration also raised two issues to be investigated. First, it is important to link the abstract Widget concept to the business engine of the project. The business engine is essentially in charge of defining flexible business models for services (including prepaid, postpaid, and promotions) within the SERVERY marketplace. Thus, the abstract Widget concept should interact with the business engine in order to retrieve the services that could be used by the user (i.e. free services, or those the user is already

[40] Open Mashups Studio, https://addons.mozilla.org/en-US/firefox/addon/7198/, accessed on March 9th, 2010.

subscribed to). Second, the price of a service is not usually a fixed value; instead it varies from a user and another (subscription option), and from a period to another (promotions). Consequently, in practice, it is not conceivable to define selection rules that refer to the service price as a fixed parameter; instead, it should be considered as a dynamic parameter which should be computed by the business engine.

4 Others (Orange Labs Internal Projects)

Several other demonstrations related to the Widget Combination capability have been conducted within Orange Labs. Some of them have been followed up with its integration. Thus, two projects have decided to integrate the Widget Combination capability to their frontend framework. The first one is “IP Accueil”; an Orange Labs internal project which is in charge of maintaining and evolving the Orange Multicanal Contact Centre product. Sold to companies, this product is an end-to-end platform (including a network infrastructure and software applications) for contact centres. The Widget Combination capability is used at the frontend level of the platform; a Widget aggregator used by the contact centre agents for achieving the business processes related to customer calls. The Widget Combination capability is then in charge of composing the Widgets between them to automate as much as possible the business processes.

The second project where the Widget Combination capability will be used is an ongoing research project. It aims to design and implement a unified messaging application, where users do not care about the medium through which the message is carried out; instead, it is the application which decides the best one to use. In this ongoing project, the team seeks to enrich the frontend framework with a draggable area, where users can add and remove services implemented as Widgets. The draggable area integrates the Widget Combination capability in order to enable communication between Widgets to enhance the user experience. In addition to the Widget Combination capability, the integration of the abstract Widget concept is under discussion. This would enable users to choose the rules to apply when selecting the best medium to use when transmitting a message.

5 Conclusions

In this Chapter we detailed the different experimentations and demonstrations we made internally in Orange Labs, or externally in the SERVERY European project. In addition to the positive results (the adoption of the WOA in different operational and research projects), two lessons were learnt. First, some users might need more integration at the UI level. While the functionalities of the Widgets are well integrated with each other, their user interfaces remain displayed as independent blocks. For instance, when combining the telephony Widget and the directory Widget (searching the caller on the directory), it could be more user friendly to display the name and the picture of a caller directly within the telephony Widget;

instead of searching the caller in a separate UI. The approach we advocate to tackle this issue is to define a Widget as an association of several UIs with a business logic. Each UI corresponds to a display type (e.g. display in the menu, display in another Widget, reduced display…etc). However, this solution is not yet investigated.

Second, additional research work is needed to use the abstract Widget concept in practice. Two important issues need to be investigated: which business model to apply? And how different pricing models could be integrated in the service selection process?

Conclusions and Future Research Directions

The main contribution of this thesis is the definition of a new service-oriented paradigm based on Widgets: the Widget Oriented Architecture (WOA) paradigm. The WOA provides integrators (developers or users) different user centric mechanisms to reuse Widgets in the creation of more sophisticated services.

Basically, the WOA paradigm is driven by two main principles: the development of services as Widgets (this requires the implementation of the UI, its semantic typing, and the creation of a description file), and the composition at the UI level. Theses two principles provide more user centricity in the process of reusing existing services.

Based on these two principles, we have designed a new Widget aggregator enriched with three Widget reusability mechanisms: the API-based reuse of Widgets, the semantic and automatic reuse of Widgets, and the process-based reuse of Widgets. The API-based reuse of Widgets is characterized by providing a set of client-side functions for developers to discover and reuse Widgets functionalities at runtime. The semantic and automatic reuse of Widgets is characterized by automatically detecting semantic matching between Widgets, and combining them. Finally, the process-based reuse of Widgets provides the capability of creating and executing a chaining of Widgets based on a process definition.

All these three mechanisms can be extended with three other mechanisms to enhance even more their user centricity: the abstract service based reuse, the cross-device based reuse, and the unstructured data based reuse. The abstract service based reuse provides the capability of invoking abstract Widgets (functionalities and a set of selection rules), which are linked at runtime to the best real Widget to execute according to rules specified by the user himself. The cross-device based reuse enables the extension of the reusability scope into Widgets loaded on different devices. This enables composition mechanisms to consider all Widgets loaded on all devices of the user. Finally, the unstructured data based reuse enables the extension of the reuse scope into data that are not declared (resp. not formatted) by providers (resp. developers) at the publication (resp. development) of the Widget. This is especially useful when considering user generated content and communication services, where significant unstructured data are generated and exchanged between users, but can not be used as input in a service composition.

In this thesis, we applied the different mechanisms in two SOA application fields, namely service composition and business process management; Figure 110 summarizes the benefits brought by each mechanism to each SOA application field.

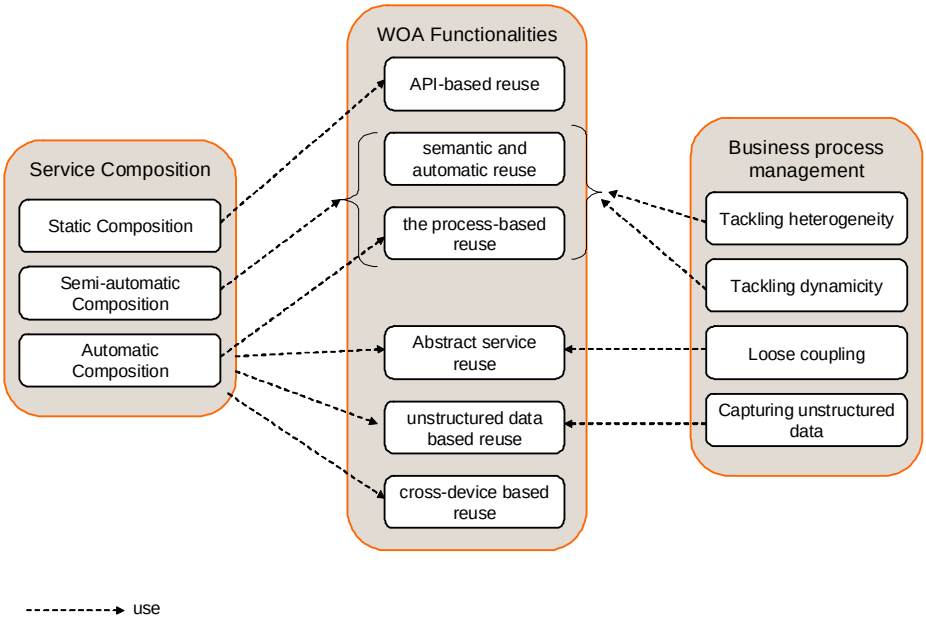


Figure 110: Contributions Summary.

The mechanisms introduced by the WOA benefit both service composition and business process management fields:

- First, the API-based reuse of Widgets provides user centricity to static service composition. It enables developers to create composite services that are customizable by users. This is characterized by anticipating a set of links between Widgets; links which are activated only when those Widgets are present in the same environment. As a consequence, the user can easily personalize the composite service by loading only the Widgets he needs/wants to the same environment.

- Second, the semantic and automatic reuse, associated to the process-based reuse, enhances significantly the intuitiveness of the process of composing services. In addition, it tackles the heterogeneity and dynamicity of business processes. Regarding service composition, the two mechanisms enable even ordinary users to create their own chaining of Widgets. This is enabled through three steps. In the first step, the platform detects semantic matching between Widgets that are loaded on the same environment, creates links between them, and creates the corresponding composite service definition (flowchart). Then, through UI elements, the user can modify the

composite service definition (deleting and modifying different links). Finally, the process-based reuse controls the execution of the new definition of the composite service. This capability of composing services provided to users enables to tackle the heterogeneity and the dynamicity of business processes. We have defined a new method for modelling and automating business processes using the WOA. This is characterized by splitting a business process into two parts, a common part and a user-dependent part. The former, modelled by business analysts and automated by developers, includes activities that are usually mandatory in an organization and/or common to a significant population of users. The latter, which is usually heterogeneous and dynamic (as it depends on each user needs and habits), is automated by the users themselves using the composition facilities provided by the WOA. This method of managing business processes benefits both business analysts and users. Business analysts will not consider the processes details related to specific users, and thus limit the number of processes to model and automate. And users are able to automate the details related to their own needs and habits.

- Third, the abstract service based reuse, brings two features to service composition and business process management: automatic adaption and loose coupling. Indeed, as service integrators (users and developers) will define composite services (resp. business processes) based on abstract Widgets (which are part of the platform), instead of real Widgets, modifications on the Widgets provider level would not affect the composite services (resp. business processes).

- Fourth, the unstructured data based reuse introduces a new composition pattern to the service composition field, and enables the automation of business processes based on the content (e.g. information included within an email content, or IM discussion). This functionality enables users to define composite services where unstructured data are first extracted from a source Widget; and then composed with another Widget. This feature is pertinent especially when considering the proliferation of user generated content and communication services, where a huge amount of data is generated by and exchanged between users. These data are not formatted at the development of the service and consequently not declared at its publication time. Consequently, they are not considered in current composition tools, especially those addressed for users.

- Fifth, the cross-device based reuse introduces a new feature to user service composition field. It enables the creation of a composition of Widgets loaded on different devices; a feature which is not enabled in current composition tools, especially those addressed for users.

Compared to SOA, which is conceived essentially to respond to developer needs in term of reusability, abstraction, and integration, WOA is built from the user perspective. The functionalities we

detailed are all initially built to respond to the needs of users. The API-based reuse functionality responds to the need of personalization of a composite service; the semantic and automatic reuse, associated to the Process-based reuse, responds to the need of creating composite services by users; the abstract service based reuse responds to the need of service selection according to users specific rules; the unstructured data based reuse responds to the proliferation of communication services and user generated content; finally, the cross-device based reuse responds to the proliferation of user devices and the need to mashup services loaded on different devices. This design methodology is inline with Web 2.0 best practice: user centred design, user participation in content creation, user self-service, and rich user interfaces.

In this thesis, we introduced the WOA paradigm. By contrast to SOA which is conceived for developers, the WOA is conceived for users. As a consequence, WOA and SOA are more complementary than concurrent. For instance, WOA based composition would never be as rich as a composition made by a developer using programming (e.g. Java) or scripting (e.g. BPEL) languages. Similarly, SOA based composition would never be as intuitive and user centric as WOA based composition. Therefore, it is important to use both approaches, and have SOA and WOA layers, in a given service environment. The approach we advocate can be summarized in the following items:

- developers create web services (SOA),
- developers create the corresponding Widgets (SOA and WOA),
- developers compose web services, create the corresponding Widget, and optionally use static composition of Widgets (SOA and WOA),
- users compose Widgets (WOA).

The mechanisms introduced in this thesis highlight new opportunities to investigate in the future, namely semantic expressiveness and combination with the users intelligence, learning from user-dependent parts of processes to sketch out automatically new business processes, and finally, unstructured data based composition in the context of multimedia content.

In the semantic field, a significant research work has been conducted in conjunction with service composition. The goal is to use highly expressive semantic and semantic reasoning to automatically compose services to reach a given goal. In this thesis we have adopted another approach. We describe our data using microformats, which define the format and the type of data, but not their meaning (semantic) in the context of a given domain. As a consequence, automatic composition of Widgets loaded to the same environment may generate links which do not have any meaning for the user. But first, the probability of generating such link is reduced as the user would likely load only Widgets of the same business domain;

and second, even if such link has been created, the user can delete it as we previously detailed. So, the lack of semantic reasoning has been compensated by the user intelligence. As a future direction, we believe that it is worthwhile to go further in this idea, and design and experiment a learning mechanism where meaningful relationships between concepts can be derived from the association of Microformats and user intelligence. More precisely, it would be pertinent to derive the relationships between the concepts from the links that are defined by users between Widgets.

In BPM field, the discovery of business processes of an organization remains a challenge. One approach for doing so is to capture the users' habits and practices. This is usually performed by analyzing (manually or automatically) the logs of different software applications that the users use. However, while such log includes the information about the usage of the application itself, it does not provide centralized information about the interactions, made manually by the users, with other applications; now, such information is pertinent in the context of a business process, which is a succession of activities, and thus likely to be a succession of applications with data exchange between them. The WOA we proposed captures such information. It captures the interaction between the different Widgets, when they are automated by the user. Therefore, it should be investigated how we can make use of such information to capture the user processes, and to link them with processes of other users in order to sketch out the company business processes (using for example Business Process Management Notation (BPMN)).

Finally, we believe that it is worth to investigating the unstructured data based composition in the context of multimedia content. Indeed, in this thesis we proposed an architecture and a proof of concept for enabling users to compose services based on unstructured data, but it is currently limited to text based data. This motivates a deeper investigation in this field, and it would be indeed interesting to consider multimedia content (audio, video, picture…etc), to enable users to create composite services that first capture useful data within the content, and then compose it with other services (e.g. capturing postal addresses exchanged during a call session, and composing them with a Map service; or capturing human faces in a souvenir movie and composing them with contact list service). The main issue is the capture of useful data, which is much more difficult in multimedia content. In addition, it must be studied the presentation aspect from one hand, and the privacy concerns from another hand.

References

- [Aguilar-Savén, 2004] Aguilar-Savén, R.S., 2004. Business process modelling: Review and framework. International Journal of Production Economics. Volume 90, no2, pp129-149.
- [Akram, 2005] Akram, A., Chohan, D., Wang, X. D., Yang, X., and Allan, R., 2005. A Service Oriented Architecture for Portals Using Portlets. In UK e-Science All Hands Conference. AHM 2005. Nottingham, UK. 2005.
- [Andrews, 2003] Andrews, T., et al. 2003. Business Process Execution Language for Web Services. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf.
- [Armbrust, 2009] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., and Zaharia, M. 2009. Above the clouds: A Berkeley view of Cloud computing. Technical report UCB/EECS-2009-28. Electrical Engineering and Computer Sciences, University of California at Berkeley. Berkeley, USA. February 2009.
- [Arsanjani, 2007] Arsanjani, A., Zhang, L., Ellis, M., Allam, A., Channabasavaiah, K. 2007. S3: A Service-Oriented Reference Architecture. IT Professional. Volume 9, no3, pp10-17. May-June 2007.
- [Beaty, 2009] Beaty, K., Kochut, A., Shaikh, H. 2009. Desktop to cloud transformation planning. In proceedings of the IEEE International Symposium on Parallel & Distributed Processing. IPDPS 2009. IEEE Computer Society, Washington, DC, pp1-8. May 2009.
- [Bellas, 2004] Bellas, F. 2004. Standards for Second-Generation Portals. IEEE Internet Computing. Volume 8, no2, pp54-60. March 2004.
- [Berners-Lee, 1998] Berners-Lee, T. 1998. Semantic web road map. Available at http://www.w3.org/DesignIssues/Semantic.html, accessed on June 14th, 2010.
- [Bertin, 2009] Bertin, E. 2009. Architecture des services de communication dans un contexte de convergence. PhD thesis.
- [Bharati, 1995] Bharati, A., Chaitanya, V., Sangal, R. 1995. Natural Language Processing: A Paninian Perspective. Prentice-Hall of India 1995. Comput. Linguist. Volume 21, no3, pp419-421. September, 1995.
- [Bichler, 2006] Bichler, M. and Lin, K. 2006. Service-Oriented Computing. Computer. Volume 39, no3, pp99-101. March 2006.
- [Bieber, 2001] Bieber, G. 2001. Introduction to Service-Oriented Programming. Online white paper. Available at

http://www.openwings.org/download/specs/service%20oriented%20programming.pdf, accessed on June 16th, 2010.

- [Casati, 2007] Casati, F. 2007. Service-oriented computing. SIGWEB Newsletter. Winter 2007.
- [Cordier, 2006] Cordier, C., Carrez, F., Van Kranenburg, H., Licciardi, C., Van der Meer , J., Spedalieri, A., Le Rouzic, J. P., Zoric, J. 2006. Addressing the Challenges of Beyond 3G Service Delivery: the SPICE Service Platform. In the 6th Workshop on Applications and Services in Wireless Networks. ASWN 2006. Berlin, 2006.
- [Cremene, 2009] Cremene, M., Tigli, J. Y., Lavirotte, S., Pop, F. C., Riveill, M., Rey, G. 2009. Service Composition Based on Natural Language Requests. In proceedings of the 2009 IEEE International Conference on Services Computing. SCC 2009. IEEE Computer Society, Washington, DC, pp486-489. September 2009.
- [Cumberlidge, 2007] Cumberlidge, M. 2007. Business Process Management with J Boss jBPM. Packt Publishing, 2007.
- [Cox, 1986] Cox, B.J. 1986. Object oriented programming: an evolutionary approach. In Addison-Wesley Longman Publishing Co., Inc. Boston, MA, 1986.
- [Dahl, 1968] Dahl, O. 1968 SIMULA 67 Common Base Language. Norwegian Computing Center. Publication.
- [Díaz, 2007] Díaz, S.P., Paz, I. 2007. Providing Personalized Mashups Within the Context of Existing Web Applications. Proceedings of the 8th international Conference on Web information Systems Engineering. WISE 2007. Lecture Notes in Computer Science. Volume 4831, pp493-502.
- [Díaz, 2008] Díaz, O., Irastorza, A., Sánchez Cuadrado, J., and Alonso, L. M. 2008. From page-centric to portlet-centric Web development: Easing the transition using MDD. Information and Software Technology. Volume 50, no12, pp1210-1231. November, 2008.
- [De Deugd, 2006] de Deugd, S., Carroll, R., Kelly, K.E., Millett, B., Ricker, J. 2006. SODA: Service Oriented Device Architecture. IEEE Pervasive Computing. Volume 5, no3, pp94-96. July-September, 2006.
- [Du, 2006] Du, Z., Huai, J., and Liu, Y. 2006. Ad-UDDI: An active and distributed service registry. In proceedings of the 6th VLDB International Workshop on Technologies for E-Services. Lecture Notes in Computer Science. Volume 3811, pp58–71.
- [Ennals, 2007a] Ennals, R., Gay, D. 2007. User-friendly functional programming for web mashups. In Proceedings of the 12th ACM SIGPLAN international Conference on Functional Programming. ICFP '07. ACM, New York, NY, pp223-234. Freiburg, Germany. October, 2007.

- [Ennals, 2007b] Ennals, R. J., Garofalakis, M. N. 2007. MashMaker: mashups for the masses. In Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data (Beijing, China, June 11 - 14, 2007). SIGMOD '07. ACM, New York, NY, pp1116-1118.
- [Erl, 2007] Erl, T. 2007. Soa: Principles of Service Design. First. Prentice Hall Press.
- [Fielding, 2000] Fielding, R.T. 2000. Architectural Styles and the Design of Network-based Software Architectures. thesis dissertation.
- [Grimes, 1997] Grimes, R., Grimes, D. R. 1997. Professional Dcom Programming. Wrox Press Ltd.
- [Goldberg, 1976] Goldberg A. and Kay A. 1976. SMALLTALK-72 Instruction Manual. Technical Report SSL-76-6, Xerox Palo Alto Research Center. Palo Alto, California.
- [Hammer, 1993] Hammer, M., Champy, J. 1993. Reengineering the Corporation: A Manifesto for Business Revolution. Harper Business, New York, NY.
- [Huhns, 2005] Huhns, M. N., Singh, M. P. 2005. Service-Oriented Computing: Key Concepts and Principles. IEEE Internet Computing. Volume 9, no 1, pp75-81. January 2005.
- [Ibrahim, 2009] Ibrahim, N., Le Mouel, F.L. 2009. A Survey on Service Composition Middleware in Pervasive Environments. International Journal of Computer Science Issues. Volume 1, pp1-12. August 2009.
- [IETF, 2001] RFC 3050. 2001. Common Gateway Interface for SIP.
- [Khare, 2006] Khare, R., Çelik, T. 2006. Microformats: a pragmatic path to the semantic web. In Proceedings of the 15th international Conference on World Wide Web. WWW '06. ACM, New York, NY, pp865-866. Edinburgh, Scotland. May, 2006.
- [Ko, 2009] Ko, R. K. 2009. A computer scientist's introductory guide to business process management (BPM). Crossroads. Vol.15, no.4, pp.11-18. June 2009.
- [Ku, 1994] Ku, B.S. 1994. A reuse-driven approach for rapid telephone service creation. In the proceedings of the third International Conference on Software Reuse: Advances in Software Reusability. vol., no., pp.64-72. November, 1994.
- [Lawton, 2008] Lawton, G. 2008. Moving the OS to the Web. Computer. vol.41, no.3, pp.16-19. March 2008.
- [Lécué, 2006] Lécué, F., Léger, A. 2006. Semantic Web Service Composition Based on a Closed World Assumption. In Proceedings of the European Conference on Web Services. ECOWS '06. IEEE Computer Society, Washington, DC, pp.233-242. December 2006.

- [Lécué, 2007] Lécué, F., Léger, A., Pires, L.F. 2007. A Framework for Dynamic Web Services Composition. In: 2nd ECOWS Workshop on Emerging Web Services Technology. WEWST07. CEUR Workshop Proceedings. Halle (Saale), Germany. November, 2007.
- [Luthria, 2009] Luthria, H., Rabhi, F. 2009. Service oriented computing in practice: an agenda for research into the factors influencing the organizational adoption of service oriented architectures. Journal of Theoretical and Applied Electronic Commerce Research. Vol.4 no.1, pp.39-56. April 2009.
- [Magedanz, 2007] Magedanz, T., Blum, N., Dutkowski, S. 2007. Evolution of SOA Concepts in Telecommunications. Computer Vol.40 no.11, pp. 46-50. November 2007.
- [Margaria, 2006] Margaria, T. and Steffen, B. 2006. Service Engineering: Linking Business and IT. Computer. Vol. 39, no. 10, pp. 45-55. Oct. 2006.
- [Margaria, 2007] Margaria, T. 2007. Service Is in the Eyes of the Beholder. Computer. Vol 40 no 11, pp.33-37. Nov. 2007.
- [McIlraith, 2003] McIlraith, S.A., Martin, D.L. 2003. Bringing semantics to Web services. IEEE Intelligent Systems. Vol.18, no.1, pp. 90- 93. Jan-Feb 2003.
- [Newcomer, 2002] Newcomer, E. 2002. Understanding Web Services: XML, Wsdl, Soap, and UDDI. In Addison, Wesley, Boston, Mass. May 2002.
- [Nitto, 2009] Nitto, E., Sassen, A., Traverso, P., and Zwegers, A. 2009. At Your Service: Service-Oriented Computing from an EU Perspective. The MIT Press.
- [OASIS, 2003] Kropp, A., Leue, C., Thompson, R. 2003. Web Services for Remote Portlets Specification. OASIS Standard. Available at http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrp-specification-1.0.pdf, accessed on March 9th, 2010.
- [OASIS, 2008] Thompson, R. 2008. Web Services for Remote Portlets Specification v2.0. OASIS Standard. Available at http://docs.oasis-open.org/wsrp/v2/wsrp-2.0-spec-os-01.html#_Toc25, accessed on March 9th, 2010.
- [OMA, 2007] OMA. 2007. Dictionary for OMA Specifications. Available at http://www.openmobilealliance.org/document/OMA-ORG-Dictionary-V2_6-20070614-A.pdf, accessed on March 9th, 2010.
- [OMG, 2008a] OMG. 2008. Common Object Request Broker Architecture (CORBA) Specification, Version 3.1. OMG specification. Available at http://www.omg.org/spec/CORBA/3.1/, accessed on March 9th, 2010.

- [OMG, 2008b] OMG. 2008. Business Process Model and Notation, V1.1. OMG specification. Available at http://www.omg.org/spec/BPMN/1.1/PDF/, accessed March 9th, 2010.
- [Papazoglou, 2003] Papazoglou, M.P. and Georgakopoulos, D. 2003. Service-Oriented Computing. Communications of the ACM. Vol.46, no10, pp.25-28. October 2003.
- [Papazoglou, 2006] Papazoglou, M.P., Traverso, P., Dustdar, P., and Leymann F. 2006. Service-Oriented Computing Research Roadmap. Technical report/vision paper on Service oriented computing European Union Information Society Technologies (IST). Available at http://infolab.uvt.nl/pub/papazogloump-2006-96.pdf, accessed on March 9th, 2010.
- [Podesta, 2008] Podesta, R., 2008. A Lightweight Inter-node Operation for UDDI Cloud. In Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops. EDOCW. IEEE Computer Society, Washington, DC, 397-400. September, 2008.
- [Pop, 2009] Pop, F., Cremene, M., Vaida, M., and Riveill, M. 2009. On-demand service composition based on natural language requests. In Proceedings of the 6th international Conference on Wireless on-Demand Network Systems and Services. IEEE Press, Piscataway, NJ, 41-44. February, 2009.
- [Saha, 2003] Saha, D., Mukherjee, A. 2003. Pervasive computing: a paradigm for the 21st century. Computer. Vol.36, no.3, pp. 25- 31. March, 2003
- [Sawyer, 2005] Sawyer, P. 2005. Service Specification State of the Art. SeCSE deliverable. Available at http://www.secse-project.eu/?page_id=80, accessed on March 9th, 2010.
- [Sire, 2009] Sire, S., Paquier, M., Vagner, A., and Bogaerts, J. 2009. A messaging API for inter-widgets communication. In Proceedings of the 18th international Conference on World Wide Web WWW '09. ACM, New York, NY, 1115-1116. Madrid, Spain. April, 2009.
- [Sillitti, 2002] Sillitti, A., Vernazza, T., and Succi, G. 2002. Service Oriented Programming: A New Paradigm of Software Reuse. In Proceedings of the 7th international Conference on Software Reuse: Methods, Techniques, and Tools. C. Gacek, Ed. Lecture Notes In Computer Science, vol. 2319. Springer-Verlag, London, 269-280. April, 2002.
- [Soriano, 2006] Soriano, J., Lizcano, D., Cañas, M., Reyes, M., and Hierro, J.J. 2007. Fostering innovation in a mashup-oriented enterprise 2.0 collaboration environment. System and Information Science Notes, International Conference on Adaptive Business Systems 2007, Vol.1, No.1, pp.62-69. Chengdu, China. July, 2007.
- [Stefan, 2008] Stefan, H. 2008. Java Portlet Specification Version 2.0. Java specification. Available at http://jcp.org/aboutJava/communityprocess/edr/jsr286/, accessed on August 22nd, 2010.

- [Steffen, 2007] Steffen, B., Margaria, T., Nagel, R., Jörges, S., and Kubczak, C. 2007. Model-driven development with the jABC. In Proceedings of the 2nd international Haifa Verification Conference on Hardware and Software, Verification and Testing. Haifa, Israel. October, 2006.
- [Sun,1999] Sun Microsystems. 1999. Java Naming and Directory Interface Application Programming Interface (JNDI API). Java specification. Available at http://www.orionserver.com/docs/specifications/jndi.pdf, accessed on August 22nd, 2010.
- [Sun, 2003] Sun Microsystems. 2003. JSR 168: Java Portlet Specification. Version 1.0. Java specification. Available at http://jcp.org/aboutJava/communityprocess/final/jsr168/index.html, accessed on July 31, 2010.
- [Sun, 2006] Sun Microsystems. 2006. JSR 220: Enterprise JavaBeans, Version 3.0 EJB Core Contracts and Requirements. Version 3.0. Java specification. Available at http://jcp.org/en/jsr/summary?id=220, accessed on August 22nd, 2010.
- [Sun, 2008] Sun Microsystems. 2003. JSR 286: Java Portlet Specification. Version 1.0. Java specification. Available at http://www.jcp.org/en/jsr/detail?id=286, July 31st, 2010.
- [Thomas, 1998] Thomas, A., Seybold, P. 1998. Enterprise JavaBeans Technology. Available at http://wiki.daimi.au.dk:8000/pca/_files/ejb_white_paper.pdf, accessed on July 31st, 2010. December 1998.
- [UWA, 2008] Borderie, X., Hodierne, F. 2008. Universal Widget API (UWA) 1.2. Netvibes Working Draft. Available at http://netvibes.org/specs/uwa/current-work/, accessed on March 9th, 2010.
- [Verma, 2005] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., and Miller, J. 2005. METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services. Inf. Technol. and Management. Vol.6, no.1, pp.17-39 January, 2005.
- [Verner, 2004] Verner, L. 2004. BPM: The Promise and the Challenge. Queue. Vol.2, no.1, pp.82-91. March, 2004.
- [Vinoski, 1997] Vinoski, S. 1997. CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Communications Magazine. Vol.35, no.2, pp.46-55. February, 1997.
- [Vo, 2006] Vo, V., Kiet, H.T., and Weinhardt, C. 2006. Corporate Portals from a Service-Oriented Perspective The CoFiPot Implementation. In Proceedings of the 8th IEEE international Conference on E-Commerce Technology and the 3rd IEEE international Conference on Enterprise

Computing, E-Commerce, and E-Services. CEC-EEE. IEEE Computer Society, Washington, DC, 32. June, 2006.

- [W3C, 2004a] Haas, H., Brown, A., 2004. Web Services Glossary. W3C Working Group Note. Available at http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/, accessed March 9th, 2010.
- [W3C, 2004b] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D., 2004. Web Services Architecture. W3C Working Group Note. Available at http://www.w3.org/TR/ws-arch/, accessed on March 9th, 2010.
- [W3C, 2004c] Bechhofer, S., et al. 2004. OWL Web Ontology Language Reference. OWL Web Ontology Language Reference. W3C Recommendation. Available at http://www.w3.org/TR/owl-ref/, accessed March 9th, 2010.
- [W3C, 2004d] Klyne, G., Carroll, J.J., McBride, B. 2004. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation. Available at http://www.w3.org/TR/rdf-concepts/, accessed on March 9th, 2010.
- [W3C, 2004e] McGuinness, D.L., Harmele, F.V. 2004. OWL Web Ontology Language Overview. W3C recommendation. Available at http://www.w3.org/TR/owl-features/, accessed March 9th, 2010.
- [W3C, 2004f] McGlashan, S., et al. Voice Extensible Markup Language (VoiceXML) Version 2.0. Available at http://www.w3.org/TR/voicexml20/, accessed on October 19th, 2010.
- [W3C, 2010] Baggia, P., Scott, M. 2010. Voice Browser Call Control: CCXML Version 1.0. Available at http://www.w3.org/TR/ccxml/, accessed on October 19th, 2010.
- [W3C, 2007] Caceres, M. 2007. Widgets 1.0 Requirements. W3C Working Draft. Available at http://www.w3.org/TR/2007/WD-widgets-reqs-20070209/, accessed March 9th, 2010.
- [Wei, 2009] Wei, Y., Sun, Z., Chen, X., Zhang, F. 2009. A service-portlet based visual paradigm for personalized convergence of information resources. In proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology. ICCSIT 2009. Vol. no. pp.119-124. August, 2009.
- [Weiss, 2005] Weiss, A. 2005. WebOS: say goodbye to desktop applications. networker. Vol.9, no.4, pp.18-26. December, 2005.
- [Wong, 2007] Wong, J. and Hong, J. I. 2007. Making mashups with marmite: towards end-user programming for the web. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '07. ACM, New York, NY, 1435-1444. San Jose, California, USA. April, 2007.

- [Yelmo, 2008] Yelmo, J.C., Del Alamo, J.M., Trapero, R., Falcarm, P., Jian Y., Cairo, B., Baladron, C. 2008. A user-centric service creation approach for Next Generation Networks. In Innovations in NGN: Future Network and Services, 2008. K-INGN 2008. First ITU-T Kaleidoscope Academic Conference. vol., no., pp.211-218. May 2008.
- [Zhang, 2007] Zhang, L.J. Zhang, J. and Cai, H. 2007. Services Computing. Springer and Tsinghua Univ. Press, July 2007.

Abbreviations

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
BPEL	Business Process Execution Language
BPEL4WS	Business Process Description Language for Web Services
BPM	Business Process Management
CDCW	Cross Device Connecting Widget
CLI	Command Line Interfaces
CLM	Causal Link Matrix
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
DaaS	Desktop as a Service
DCOM	Distributed Component Object Model
EJB	Enterprise JavaBean
GC	Grid Container
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
IDL	Interface Description Language
IM	Instant Messaging
IT	Information Technology
J2EE	Java Enterprise Edition
JBPM	Java Business Process Management
JNDI	Java Naming and Directory Interface
JPDL	JBPM Process Definition Language
JS	JavaScript
JSR	Java Specification Request
KB	Knowledge Base
LASR	Local Automatic and Semantic based Reuse

NLC	Natural Language Composer
OLE	Object Linking and Embedding
OMA	Open Mobile Alliance
OMG	Object Management Group
OOP	Object-Oriented Programming
OPUCE	Open Platform for User-centric service Creation and Execution
ORB	Object Request Broker
OS	Operating System
OWL	Ontology Web Language
RDF	Resource Description Framework
REST	Representational State Transfer
RPC	Remote Procedure Call
P2P	Peer to Peer
PaaS	Platform as a Service
PC	Personal Computer
PMC	Process Manager Component
SAAS	Software as a Service
SA-WSDL	Semantic Annotation Web Service Description Language
SeCSE	Service Centric Systems Engineering
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOP	Service-Oriented Programming
SOC	Service-Oriented Computing
SWS	Semantic Web Services
SPICE	Service Platform for Innovative Communication Environment
TTM	Time To Market
TV	Television
UDDI	Universal Description, Discovery and Integration
UGC	User Generated Content
UI	User Interface
URL	Uniform Resource Locator
UWA	Universal Widget API
W3C	World Wide Web Consortium

WADL	Web Application Description Language
WC	Widget Container
WIMP	Windows, Icons, Menus, and Pointer
WOA	Widget-Oriented Architecture
WSA	Web Service Architecture
WS-BPEL	Web Services Business Process Description Language
WSDL	Web Service Description Language
WSRP	Web Service for Remote Portlets
XaaS	Everything as a Service
XML	eXtensible Markup Language