Le Développement Agile de Services de Télécommunication Intégrés via des techniques d'ingénierie des modèles.

Mariano Belaunde

N° d'ordre : UEB/UR1 01-4317 **ANNÉE 2011**

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : (Nom de la mention)

Ecole doctorale MATISSE

présentée par

Mariano Belaunde

préparée à l'unité de recherche UR1
Mathématiques et Informatique

Intitulé de la thèse:

Le Développement Agile de Services de Télécommunication Intégrés via des techniques d'ingénierie des modèles.

Thèse soutenue à RENNES
le 20 Janvier 2011
devant le jury composé de :

Pierre-Alain Muller
Professeur à l'Université de Haute Alsace
rapporteur
Marten van Sinderen
Professeur à l'Université de Twente
rapporteur
Marie-Pierre Gervais
Professeur à l'Université de Nanterre
rapporteur
Jean-Louis Pazat
Professeur à l'INSA Rennes
Jean Marc JEZEQUEL
Professeur - Directeur de Thèse

Abstract (In french)

Titre: Développement Agile de Services de Télécommunication Intégrés via des techniques d'ingénierie des modèles.

Pour devenir ou rester compétitif, un opérateur télécom doit constamment enrichir ou adapter son offre de services. Cette recherche permanente d'innovation implique de rendre *agiles* les processus de création de service. Par agilité nous entendons non seulement la capacité à mettre rapidement sur le marché de nouvelles idées de service mais également de s'assurer de leur évolution dans un environnement technologique changeant.

Dans ce mémoire de thèse nous défendons l'idée qu'une utilisation *pragmatique* et combinée des principes du SOA avec les technologies d'ingénierie des modèles peut être un facteur clef pour l'optimisation du processus de création de services et pour répondre aux exigences de l'agilité. L'approche que nous recommandons dans le cas des services de télécommunications que nous avons étudié (les services composites intégrés et les services vocaux) c'est d'abord d'utiliser des langages dédiés (DSL) graphiques et/ou textuels pour la spécification de haut niveau des services, ensuite d'exploiter ces spécifications dans des environnements de création et d'exécution (*frameworks*) *orientés modèles* supportant nativement le DSL, afin de permettre le test et la simulation au plus tôt des fonctionnalités du service via des itérations rapides. Enfin une automatisation importante du déploiement vers les plates-formes de production (serveurs d'application) et les terminaux mobiles (code client) via le développement de transformateurs dédiés.

Le travail présenté dans cette thèse est validé par plusieurs expérimentations et démonstrations portant sur des services vocaux et des services de télécommunication *intégrés* composites (exploitant des ressources télécom et des facilités issues de l'industrie informatique).

Mots-clés : MDA, SOA, DSL, Orchestration

Abstract (in English)

Title: Agile development of integrated telecommunication services using model engineering.

To become or remain competitive telecom operators continuously need to enrich and adapt their service offers. In order to bring such permanent innovation it is necessary to take care of *agility* in the process of service creation. Agility means not only the capacity to put quickly in the market innovative services but also the capacity to ensure their evolution taking into account technological changes and new expectations from end-users.

In this report we defend the idea that a pragmatic combination of SOA principles and model-engineering technology offers a promising basis for improving the development process of telecommunication services to match as much as possible agility requirements. The suggested approach firstly make use of domain specific languages (DSL) adapted to telecom context, secondly, relies on the exploitation of native frameworks supporting the DSLs for quick and iterative service prototyping and simulation and finally implies the usage of effective model transformation techniques to ensure portability and deployment of telecommunication services across different execution environments - such as those bring by modern smart-phones.

The report presents some application use cases validating our approach going from the development of voice-based applications to the development of composite services combining communication facilities and internet services, modelled through graphical or textual notations.

Keywords : MDA, SOA, DSL, Orchestration

Extended Summary (in French)

Le Développement Agile de Services de Télecommunication Intégrés via des techniques d'ingénierie des modèles.

Contexte

Les opérateurs télécoms se battent pour attirer de nouveaux clients et fidéliser les clients existants, en enrichissant leur offre de services et en les adaptant. Afin d'apporter ces innovations rapidement, il est nécessaire de prendre soin de l'agilité dans le processus de création de services.

Par l'agilité nous devons comprendre au moins deux choses: d'un côté la capacité de mettre sur le marché aussi vite que possible des services innovants, à un prix raisonnable, et dun l'autre côté, probablement le plus important, la capacité de faire évoluer les services existants de sorte qu'ils soient adaptées aux nouvelles attentes, parfois imprévisibles des utilisateurs finaux.

L'agilité dans la création de service est devenue un enjeu majeur pour les opérateurs télécoms pour rester compétitif dans le monde changeant d'aujourd'hui car c'est la richesse et la précision de leur offre qui fera la différence face aux compétiteurs. Cela est particulièrement vrai dans un contexte où les nouveaux entrants de l'industrie de l'information cherchent à profiter de la convergence informatique/ télécommunications pour disputer aux opérateurs de télécommunications traditionnels la manne de revenus de la téléphonie fixe et mobile.

L'agilité dans la création de services de télécommunications a également des répercussions sur les fournisseurs tiers de services qui sont intéressés à combiner les fonctions clés de communication offerts par les opérateurs avec leurs propres blocs fonctionnels apportant de la valeur ajoutée sur les services. En revanche, pour un opérateur, il est important de capter la plus grande communauté de développeurs de logiciels tiers afin qu'ils intègrent dans leurs applications l'utilisation de leur propre infrastructure de réseau (comme un envoie de SMS payant). La co-innovation - des partenaires qui acceptent de travailler ensemble pour partager les avantages d'une innovation - entre les opérateurs et fournisseurs de services tiers rend nécessaire pour les opérateurs de fournir des moyens contrôlés à accéder aux ressources réseau, qui, traditionnellement, étaient fermement verrouillées. Cela se traduit aujourd'hui par la disponibilité des API ouvertes publiées par les opérateurs (pour la gestion des appels, envoi de messages carnet d'adresse, et ainsi de suite) comme l'initiative d'Orange Partners (voir http://www.orangepartner.com). Des API de développement pour permettre aux développeurs d'accéder aux ressources réseau c'est une première étape. De notre point de vue, l'étape suivante pour faciliter la co-innovation dans le développement de services est de se mettre d'accord sur l'utilisation des formalismes de modélisation de haut niveau pour la définition des *services composites*, qui sont des services qui regroupent des blocs préexistants provenant soit du domaine Telco ou du domaine de l'informatique et de l'internet. Une partie de notre contribution a été consacrée à la définition de ce type de formalisme intégrant des fonctionnalités orientées télécom.

Dans ce travail, nous ferons souvent référence à des modèles. Un modèle est une représentation simplifiée, généralement abstraite d'un processus, un système, destiné à le décrire, l'expliquer ou de prévoir son comportement [dict01]. Dans notre contexte spécifique, un modèle est une spécification d'un service de télécommunication: elle décrit les fonctions offertes, la structure des données manipulées et son comportement. Il est formalisé sous la forme d'une structure de données lisibles par une machine pour permettre de nouveaux calculs et des raisonnements.

Afin d'améliorer les délais de commercialisation, les concepteurs de services doivent d'abord penser à la réutilisation de composants déjà déployés. Des questions de conception typiques sont: comment puis-je partitionner mon application pour une réutilisation optimale des modules? Quels sont les éléments existants que je peux utiliser? Toutefois, la réutilisation de composants comporte des risques: parfois intégrer un composant préexistant est plus coûteux que de redévelopper un à partir de zéro. Cela peut se produire si le composant intégré introduit des dépendances qui sont difficiles à maintenir. À cet égard, la SOA - Service Oriented Architecture - qui mettent l'accent sur le couplage léger entre les éléments potentiellement distribués, fournit un cadre attrayant pour réaliser l'agilité: la séparation entre réalisation d'un service et la publication de ses interfaces (l'API), évite les difficultés telles que la contrainte de développer des composants en utilisant un langage de programmation unique ou le problème de l'importation de bibliothèques incompatibles dans le même espace de développement. Le SOA attire de plus en plus l'attention de l'industrie des télécommunications. Le fait que les moyens de communication comme l'envoi de SMS et de contrôle des appels soit maintenant accessibles en utilisant les services web facilite significativement l'intégration de ces facilités par des fournisseurs tiers: pas besoin d'être un expert dans le domaine des télécommunications pour utiliser les fonctionnalités offertes, du moment que le développeur a accès à la documentation pour l'accès et le paramétrage du composant. *Last but not least*, dans le processus visant à rendre agile la création de services, la normalisation est appelée à jouer un rôle très important en tant que facilitateur de l'intégration: la capacité d'échange de composants grâce à une certaine uniformité dans les formats et les conventions pour représenter la logique et la données simplifie l'évolution fonctionnelle et la maintenance des services développés.

L'autre aspect de l'agilité dans la création de services est la capacité de déployer des services sur des environnements d'exécution différents. Cela vaut pour le code d'application s'exécution du côté serveur ainsi que le code de l'application s'exécutant du côté du terminal (smartphones, télévision relié à Internet et ainsi de suite) pour laquelle il y a, de nos jours, une hétérogénéité incroyable de plates-formes disponibles (Symbian, iPhone, Android et ainsi de suite).

La nécessité de supporter l'hétérogénéité des plates-formes aussi émerge de la nécessité de portabilité sur différents terminaux que les utilisateurs finaux peuvent souhaiter: si je veux que mon service soit utilisé par l'ensemble de la communauté, je vais fournir le logiciel client pour la plupart des terminaux téléphoniques populaires qui existent à un moment donné. L'utilisation de modèles combinés avec des générateurs de code différents - un pour chaque plate-forme cible - est une façon de réduire des coûts. Dans ce cas de figure, nous allons d'abord créer des spécifications partielles ou complètes des

services sous la forme de modèles et ensuite nous en déduirons automatiquement des implantations sur différentes cibles. La langage SPATEL que nous avons défini dans le cadre de notre contribution est un exemple de formalisme de haut niveau qui simplifient le développement des services déployables sur des plateformes différentes.

Le support multi-plate-forme peut aussi être motivé par le fait que la technologie évolue rapidement, ce qui peut amener à des changements dans l'infrastructure d'exécution des services, comme par exemple pour résoudre les problèmes d'équilibrage de charge lorsque le service devient très utilisé du fait de sa popularité.

À ce stade, les techniques d'ingénierie des modèles entrent sur scène: tout d'abord comme un moyen conceptuel pour l'organisation de la séparation des préoccupations (fonctionnel et technique), puis comme un outil de productivité grâce à l'automatisation de la production de code et de procédures de test, via des techniques de transformations de modèles.

Synthèse de la contribution

La thèse défendue dans ce rapport est qu'une combinaison appropriée des deux paradigmes, l'architecture orientée services (SOA) d'une part, et le Model Driven Engineering (MDE) d'autre part, est le fondement pour offrir l'agilité dans le processus de développement des services de télécommunications, c'est-à-dire, le développement rapide de nouveaux services ou leur évolution, en prenant en considération les contraintes typiques des opérateur de télécommunications comme l'hétérogénéité des plateformes d'exécution et la portabilité vers différents types de terminaux.

Plus précisément, pour gagner en agilité, nous défendons la pertinence d'une approche fondée sur trois points:

Tout d'abord, l'utilisation d'un formalisme de haut niveau exécutable pour décrire des services, qui soit adaptée à la complexité inhérente des services de télécommunication (de longue durée, basé sur des événements asynchrones, la multi-modalité, les problèmes de sécurité et ainsi de suite) et en ligne avec la philosophie de couplage faible de l'architecture SOA. Un exemple de langage dédié est la langue SPATEL que nous avons défini et mis en œuvre dans nos expériences. Il est basé sur les machines d'état comme paradigme d'exécution.

Deuxièmement, pour une mise en œuvre efficace, l'utilisation d'un framework d' exécution natif supportant le plus directement possible le langage de spécification de haut niveau, pour servir non seulement comme un environnement de simulation, mais aussi comme un environnement «par défaut» d'exécution. En plus de ce framework, le concepteur/développeur sera amené à itérer entre les différentes phases de développement du cycle de vie de service, en appliquant les recettes du prototypage rapide sur la base de modèles exécutables. Dans notre travail, le framework d'exécution natif pour exécuter des descriptions de services en SPATEL s'appelle SPATEL Engine. Il offre des fonctionnalités avancées pour gérer la variabilité de la mise en œuvre de services.

Troisièmement, le développement d'une série de transformateurs pour faire face au problème de déploiement dans les différentes plates-formes d'exécution cible et la portabilité dans différents types de terminaux téléphoniques. Des composants génériques pour le développement dirigé par les modèles peuvent avoir un impact important en tant que facilitateurs et accélérateurs de l'activité de développement de logiciels. On pense notamment au framework de méta-modélisation et au langage de transformation de modèles. Dans nos expériences nous avons utilisé PyMOF, une implantation en langage Python du standard MOF ainsi qu'un moteur d'exécution QVT Opérationel (SmartQVT) - langage standardisée à la définition de laquelle nous avons fortement contribué. Ces deux technologies, utilisées séparément ou en combinaison ont contribué à développer efficacement les transformateurs qui ont été nécessaires pour construire le cadre du service agile création.

Pour terminer, dans cette thèse nous affirmons la nécessité de combiner SOA et MDA d'une manière *judicieuse* et *pragmatique*. Malgré ses avantages potentiels, une exploitant trop dogmatique de l'ingénierie de modèles peut conduire à la construction d'un environnement de développement de services qui est trop complexe pour être efficacement maintenu et qu'à la fin apportera peu d'avantages en termes d'agilité aux concepteurs et développeurs de service.

Synthèse de l'Etat de l'Art

Nous avons étudié l'état des pratiques actuelles en ingénierie des modèles et ingénierie des services ainsi qu'un certain nombre de travaux de recherche autour de la combinaison des technologies MDA et SOA pour développer des services composites intégrés de télécommunication et des services vocaux interactifs.

Nous observons que dans les projets de recherche beaucoup d'effort est mis dans une utilisation spécialisée de UML pour capturer les spécifications comportementales des services, et ensuite traduite automatiquement vers les formalismes exécutables SOA tels que WSDL et BPEL. Différentes approches émergent:

Dans [Bauer04] les diagrammes séquence sont utilisés (entre autres diagrammes) pour dériver des orchestrations BPEL. Dans [Dumez08] des diagrammes d'activité sont exploités pour exprimer des processus composites en OWL-S et pour générer du BPEL. Dans [Zhu09] les diagrammes activité UML sont également utilisés pour exprimer processus composite OWL-S, mais la cible est un langage généré afin de vérifier la cohérence des spécifications.

Dans [Belouada10] des diagrammes BPMN représentent la logique de service et en parallèle des extensions aux diagrammes de classes UML sont proposées pour insérer des annotations sémantiques.

Le travail sur le schéma de workflow dans [Gronmo04] se concentre sur les alternatives de conception lors de l'utilisation des diagrammes d'activité pour représenter le

comportement. Enfin [Lin09], met l'accent sur l'interface graphique et le comportement défini à l'aide de diagrammes d'activité pour générer des applications VoiceXML avec l'interactivité en fonction.

Dans le domaine de la normalisation, nous voyons que, grâce à SoaML, un progrès important a été réalisé en proposant un formalisme standardisé pour décrire les aspects statiques d'un service (la structure des contrats, des interfaces, des composants de réalisation), tout en restant non prescriptif pour la partie comportementale.

De notre point de vue, les aspects suivants ne sont pas assez étudiés dans la recherche actuelle en matière de développement de services assistée par du MDA:

- La capacité à modéliser d'une manière intégrée les différents aspects du développement de services de télécommunications: cela comprend non seulement des interfaces de programmation et de comportement avec ou sans de la communication asynchrone, mais aussi, la définition sémantique, les propriétés non-fonctionnelles et la définition de l'interaction avec l'utilisateur. En particulier, l'inclusion de l'interaction vocale dans la conception de «ordinaire» des services n'est généralement pas considérée (services vocaux interactifs représentent une catégorie très spécifique de services). Les projets de recherche ont tendance à proposer des solutions qui mettent l'accent sur un aspect spécifique.

- Rôle des frameworks d'exécution de type MDA pour atteindre l'agilité. La plupart des travaux de recherche traitant de MDA mettent l'accent dans la définition de formalismes appropriés pour définir des spécifications de services indépendants aux plate-forme d'exécution. Cependant, la résolution des problèmes tels que la substitution d'un composant de service par un autre équivalent ou la construction de services sensibles au contexte, dépendent dans une large mesure des caractéristiques des environnements d'exécution. En d'autres termes, la question qui se pose est: quelle conséquence a l'introduction du MDA dans la conception des environnements d'exécution modernes? Ceci s'oppose à la vision traditionnelle dans laquelle le MDA tente simplement de *mapper* des middlewares existants, sans que ces middleware aient une visibilité et une prise sur les modèles.

Approche pour réaliser l'agilité dans les services de télécommunications

Les aspects motivation entre membres d'une équipe de développement et plus globalement les aspects de gestion de projet, comme souligné par les auteurs du manifeste agile [Fowler01] sont d'une importance primordiale pour mettre en place des processus de développement agiles. Notre focus sera toutefois sur les moyens techniques que nous pouvons mettre en place pour aider le processus de développement de services de télécommunications d'être aussi productifs et adaptables que possible. Dit d'une autre manière, notre attention porte sur l'outillage - pas l'équipe -, entendue dans son sens large, qui inclut non seulement le logiciel concret mais les abstractions sur lesquelles il est basé (formalismes, les notations, les relations de correspondances (*mappings*), les meilleures pratiques et ainsi de suite).

Pour le cas spécifique du développement de services vocaux et services de télécommunications composite, nous avons trouvé pertinent l'ajout des trois principes suivants pour réaliser l'agilité au niveau de l'outillage:

- Utilisez une ou plusieurs langages spécifiques au domaine (DSL) pour spécifier les aspects pertinents d'un service dans une mise en œuvre agnostique vis-à-vis de la réalisation.

- Utiliser des outils qui permettent une exécution immédiate du DSL dans une plate-forme de déploiement par défaut pour permettre des tests immédiats et la simulation itérative du service en cours d'élaboration.

- Utilisez des outils qui automatisent le plus possible la production et le déploiement du service dans les différentes plateformes d'exécution du côté du terminal et du côté serveur.

Suivant le modèle RUP [Kroutchen], nous nous appuierons sur un cycle de développement constitués de 4 phases principales: inception, elaboration, construction et transition.

Focus sur la phase de construction

En substance la phase de construction sera en charge de la création de la spécification détaillée des services, d'appliquer la génération de code et de réaliser la mise en œuvre complète. La phase est fortement tributaire de la disponibilité des outils de production. Le service sera formellement spécifié à l'aide d'un DSL. L'hypothèse que nous faisons est qu'un service est à peu près défini par une interface de service incluant des attributs de configuration, des opérations, où les opérations de service peut avoir ou ne pas avoir une logique explicite définie.

Cinq tâches typiques pour la phase de construction ont été identifiées, en prenant comme élément d'entrée une décomposition des fonctionnalités du services établie lors de la phase d'élaboration (non détaillée ici):

- I1: Spécification: définir l'interface de service de chaque composant, nécessaires à l'itération qui est en cours d'exécution. A ce stade, l'interface d'un composant externe (ou *ami*) peut être adapté aux besoins du nouveau service. Dans ce cas, il devient un service de médiation pour le service externe. Une motivation pour l'adaptation d'une interface existante est de la rendre aussi neutre que possible et par là de faciliter une substitution future en cas de changements dans l'environnement.

- I2: Implantation initiale : Mise en œuvre du code de chacun des composants pour la plate-forme d'exécution par défaut. En pratique, cela comprend la logique de comportement des opérations de service, éventuellement complétées par les aspects interface graphique. Dans certains cas, la mise en œuvre peut être un code *bouchon*, qui est une mise en œuvre réalisant un travail simplifié - peut-être rien du tout - par rapport au comportement final attendu. Si le comportement du service est explicitement modélisé (par exemple via une machine d'état) dans ce cas le code sera automatiquement déduit de la

spécification. Dans le cas contraite (opération opaque) un squelette de code peut être généré pour accélérer une mise en œuvre manuelle.

- I3: Simulation: Doit permettre une exécution immédiate du service à chaque fois qu'on atteint un palier stable de la spécification. La simulation se fait via la plateforme d'exécution par défaut attaché au DSL (voir I1).

- I4: Déploiement multi-cible: Tout d'abord, compléter les interfaces de service et de la logique de service avec des informations utiles pour générer, si besoin, les différentes implantations du service ou les interfaces d'accès alternatifs (par exemple la génération d'une mini-application pour chacune des plates-formes de smartphones les plus populaires: l'iPhone, Android et Nokia S60). Ensuite, effectuez la génération de code automatique et la complétion manuelle requise afin de réaliser les différentes implantations du service.

- I5: Publication: Certains composants mis en œuvre dans I2, seront promu pour la réutilisation. Typiquement, ils vont être publiés en tant que services web autonomes.

La phase de construction que nous avons décrite ci-dessus est, comme l'ensemble du processus global, itératif et incrémental. En particulier, les erreurs constatées lors de la génération de code (tâche I4) peuvent mettre en question le design effectué dans les tâches antérieurs.

Le langage SPATEL

La composition de services est devenue un sujet brûlant pour tous les acteurs des télécommunications. La possibilité pour les professionnels et, encore plus pour les utilisateurs finaux, de pouvoir composer efficacement des briques de services telecom, dépend beaucoup de la disponibilité d'outils capables de cacher la complexité pour accéder aux ressources mis à disposition par l'opérateur. De nombreuses initiatives sont actuellement lancées par les opérateurs pour ouvrir l'accès à leurs ressources réseau, comme le programme Orange Partner pour la 3ème partie des développeurs [orangepartner].

Dans cette section, nous présentons notre solution proposée, qui inclu essentiellement deux éléments: le langage dédié SPATEL et de l'environnement de création de services construit au-dessus de celui-ci, principalement via le moteur d'orchestration appelé SPATEL Engine. Nous soulignons ici l'importance d'avoir des artefacts en place (le DSL et le framework qui les exploite) afin de réaliser la vision d'agilité dans la création de services.

Le langage SPATEL permet la spécification des différents aspects d'un service tel que l'interface de service, la logique de fonctionnement du service (comme la logique d'orchestration d'un service composite), les dialogues d'interaction vocale, les éléments d'interface graphique (GUI), les annotations sémantiques (entrées, sorties, pré-conditions, effets et objectifs) et les propriétés non-fonctionnelles des services. SPATEL permet aux compositions de service d'être représentés en utilisant des machines

d'état, par conséquent, permettent la formulation d'interactions complexes dans une orchestration, éventuellement avec une communication asynchrone et une exécution à longue durée (*long-running*). Une fois que les spécifications SPATEL des services sont disponibles ils sont publiés sur un registre, ces services peuvent être découverts, sélectionnés et utilisés dans de nouvelles compositions de services. Les services composés sont typiquement distribués à travers le réseau et ne sont pas administrés par une entité unique.

Le langage SPATEL est indépendant de la plateforme dans le sens où elle permet de définir des interfaces de service et logique de service d'une manière technologiquement agnostique: aucune hypothèse n'est faite sur le moteur d'exécution utilisé et les protocoles de communications mis en œuvre pour exécuter les services décrits. L'un des objectifs recherchés du SCE construit au-dessus de SPATEL a été l'idée de permettra au concepteur-développeur de découvrir les services correspondant à un objectif particulier et être en mesure de proposer des compositions des services existants pour réaliser l'objectif. Afin d'atteindre un certain degré d'automatisation, des annotations sémantiques doivent être ajoutés à des descriptions de service. Comme la composition dynamique de services est basée sur le raisonnement sur la sémantique de service, des mécanismes sont définis pour permmetre l'annotation des composants de services répertoriés.

Le langage SPATEL est défini au moyen d'un méta-modèle [omg-mof] à partir duquel une API de programmation et de la sérialisation XML lisible par une machine sont déduits. Associé à ce métamodèle, il y a deux notations concrètes pour les utilisateurs: une notation textuelle pure et une notation graphique basée sur un profil UML [omg-uml]. Selon le type d'utilisateurs, l'une des deux notations offertes peuvent être utilisée: la version graphique est particulièrement adapté au travail collaboratif entre les concepteurs de services, mais elle peu devenir plus difficile à gérer qu'une notation textuelle compacte dans le cas de la formalisation d'une logique de services complexe.

Un service en SPATEL est d'abord essentiellement décrit comme une interface «boîte noire» qui fournit les informations dont les clients ont habituellement besoin pour collaborer avec elle. Cette interface de service déclare une liste d'pérations, une liste d'événements d'entrée et de sortie, des flux multimédias et des *effets*. Des contraintes d'utilisation tels que l'ordre des invocations d'opération peut être précisément définie par un contrat spécifié par le biais d'un diagramme de séquence UML.

En plus de la vue externe décrit ci-dessus, le langage SPATEL permet de décrire le service comme une *boîte blanche*, qui expose une spécification partielle ou complète de son comportement interne. Plus précisément, la logique d'une opération de service dans l'interface peut être définie comme une orchestration - une composition centralisée - d'autres services. A contrario des approches plus «traditionnelles» pour les services web, une opération de service dans notre contexte de télécommunication peut être de longue durée et son exécution peut être interrompue en attendant l'arrivée des notifications d'événements asynchrones. Le paradigme utilisé dans SPATEL pour supporter ce genre de comportement est la machine d'état. Les machines d'état sont particulièrement utiles pour

représenter les interactions complexes généralement utilisés dans les applications vocales ou dans les services multi-modaux dans les téléphones mobiles.

Annotations sémantiques

Le langage SPATEL fournit un mécanisme générique pour l'ajout d'annotations sémantiques sur lés éléments décrivant un service ainsi que l'ajout d'informations portant sur des caractéristiques non fonctionnelles du service. Leur principal objectif est d'aider à la découverte des services (au moment du design ou à l'exécution) et de permettre des scénarios où la composition dynamique intervient.

Les annotations sémantiques sont introduites sous la forme de références vers des concepts d'une ontologie externe, définie dans RDF [RDF-w3c] ou OWL [W3C-owl]. Une ontologie défini une taxonomie des concepts enrichie par des relations sémantiques entre les nœuds, chaque concept étant défini comme un sous-ensemble de ses parents. Des conditions portant sur les concepts et les relations peuvent être spécifiées via un formalisme formel (tel que *Description Logic* (DL)).

Une ontologie spécifique au domaine des télécommunications appelée Ontologie *Mobile* [Villalonga] a été définie et utilisée dans des descriptions de services SPATEL pour permettre la découverte automatique de services dans un processus de composition automatique. L'Ontologie Mobile est structurée en sous-ontologies couvrant différents domaines: propriétés non fonctionnelles, dispositifs d'entrées-sorties, objectifs, contexte, profil utilisateur, présence, sphère de communication, contenu et confidentialité.

En résumé, concernant le langage SPATEL, Différentes approches opportunistes sont prises pour spécifier les différents aspects d'un service: la partie vocale utilise spécialise les actions et les types d'évènements UML dans une machine d'état alors que la partie interface graphique utilise une approche représentation générique (basée sur la disponibilité de bibliothèques de widgets, qu'on peut ajouter en fonction des besoins et des plateformes cibles supportées).
Pour résumer le formalisme SPATEL intègre et unifie des concepts provenant de différentes sources (VoiceXML [w3c-vxml], l'UIT-SDL [UIT-sdl], SA-WSDL [w3c-WSDL] et UML [OMG-uml]) afin de permettre aux concepteurs de service de spécifier à un niveau d'abstraction élevé mais exécutable les différentes facettes de la description d'un service.

Le Framework SPATEL Engine

Il ne suffit pas de définir un langage ayant de bonnes caractéristiques en termes de niveau d'abstraction et d'expressivité pour rendre possible un processus de développement de service agile. Une bonne partie de l'intelligence nécessaire à l'agilité devra être placée dans les transformateurs de modèle, les générateurs de code et dans les environnements d'exécution. Une exécution immédiate des spécifications du service sont nécessaires pour mettre en place des itérations fréquentes entre spécification et implantation, comme cela est

recommandé dans presque toutes les méthodologies agiles. Le framework SPATEL Engine joue le rôle de plate-forme cible native par défaut pour l'exécution des services spécifiés dans SPATEL. Il va permettre en particulier l'exécution immédiate des spécifications.

SPATEL Engine offre tout d'abord le moteur d'exécution pour exécuter la logique de service (notamment les orchestrations) définie en SPATEL. En parallèle il fournit:

- Les générateurs de code pour produire le code exécutable à partir d'une spécification de service en SPATEL. Pour les services élémentaires des squelettes de code sont générés. Pour les services composites tout le code est généré à partir de la définition SPATEL (en, machine d'état).

- Un référentiel interne de services pour stocker le code généré (ou édité manuellement) des services élémentaires et des services composites. Les services hébergés dans ce référentiel peuvent être exécutés à distance à l'aide soit à l'aide d'interfaces de type REST [Fielding00] ou via le protocole SOAP [w3c-savon], ou encore via l'utilisation de formulaires HTML.

Les services vocaux

Les services vocaux interactifs basés sur la synthèse et la reconnaissance de la voix sont des applications spécifiques à la téléphonie qui sont conçues pour permettre aux utilisateurs finaux d'obtenir des services sans passer par un opérateur humain. La formalisation de l'interaction vocale - appelé une boîte de dialogue - se décrit généralement sous la forme d'une machine d'état qui exécute la logique de la conversation. Au sein de la machine d'état on peut invoquer du code métier. Parce que machines d'état peuvent être spécifiées et modélisées formellement, il est possible de concevoir un outil qui automatise la réalisation du service sous la forme de code exécutable.

Nous le cas particulier des services vocaux nous avons défini un langage dédié appelé VOICE et un framework natif supportant le langage appelé VoiceBench. Le langage dispose de notations graphiques et de notations textuelles pour spécifier le dialogue. La notation graphique est basée sur les machines d'état UML. Vis-à-vis de SPATEL, VOICE est davantage spécialisé: ainsi les services sont structurés en *dialogues* et non sous formes d'interfaces banalisées.

Le framework VoiceBench agrège différents composants: un éditeur de modèles, un simulateur (basé sur le langage IF [Bozga04]), un moteur d'exécution accessible en http et un générateur de tests produit à partir du simulateur.

Principes de Validation

Quels sont les avantages en termes d'agilité induits par le développement et l'utilisation d'une chaîne d'outils basés sur des modèles pour la création de services?. Quels sont leurs coûts?

Ce sont des questions typiques que nous avons à répondre pour évaluer le rendement d'investissements liés à l'utilisation du MDA pour le développement de services. Dans cette section, nous allons tenter d'énoncer quelques conclusions en relation avec certaines hypothèses.

Les trois hypothèses que nous voulons vérifier sont la suivants:

H1: L'utilisation de MDA dans les outils de création de services améliore la productivité des concepteurs et des développeurs de services

H2: la modélisation explicite de la logique de service facilite l'évolution des services, même pour des services complexes.

H3: Les outils de création de services exploitant le MDA sont difficiles à développer, mais facile à maintenir.

Il convient de souligner que les hypothèse H1 et H2 concernent les utilisateurs d'un environnement de création de services (concepteurs et développeurs de service) alors que l'hypothèse H3 concerne ceux quui sont en charge de développer l'environnement de création de services.

Nous avons développé trois expériences. La première expérience concerne le développement d'un service vocal de grande taille (un service pour accéder à un carnet d'adresse en ligne en utilisant la voix) de deux manières: avec «méthode traditionnelle», puis utilisant l'approche dirigée par les modèles. Cette expérience nous aidera à évaluer H1. L'expérience suivante concerne le développement de services composites combinant des ressources télécom avec des facilités des technologies de l'information: une planification de dîner pour des touristes ensituation de mobilité. Ces deux expériences nous aideront à évaluer H2 et indirectement H1. La troisième expérience concerne l'effort pour développer et maintenir une chaîne d'outils MDA - en fait, la chaîne d'outils VoiceBench. Cela fournira des informations pour l'hypothèse H3.

Synthèse de la validation

Une chaîne d'outil basée sur le MDA permet de faire des gains de productivité significatifs aux concepteurs et développeurs de services service (hypothèse H1). Les mesures réalisées au cours de l'étude (détaillées en annexe) ont montré qu'il était possible d'obtenir 25% de gain de productivité dans les activités de conception et 70% de gain de productivité dans les activités de mise en œuvre grâce à l'utilisation de la chaîne d'outils.

En outre, l'utilisation couplée d'un langage de haut niveau pour la spécification de services avec un environnement de services adapté au langage facilite l'agilité dans l'évolution des services composites (hypothèse H2).

Toutefois, la construction et la maintenance de chaînes d'outils orienté modèles à un coût et celui-ci est loin d'être négligeable. Selon nos mesures, le développement et la maintenance du framework VoiceBench coûtait environ ~ 1 année-personne en termes de ressources et nécessitait d'environ 0,4 années par personne pour son entretien. Hypothèse H3 n'est pas vérifiée.

Conclusions

Le secteur des télécommunications a tendance à utiliser de plus en plus les technologies qui proviennent de l'industrie de l'information. Cette évolution a en effet été accélérée avec la croissance de l'Internet. Les services de télécommunications basés sur la voix offrent une bonne illustration de cette tendance: la norme VoiceXML du W3C a permis de développer une application vocale interactive d'une manière similaire à la façon que l'on développe les applications web.

Une situation similaire se produit avec les services intégrés de télécommunications combinant des facilités de communication issues des télécom (comme la messagerie, la présence et ainsi de suite) avec des composants issus de l'internet (traduction, météo, flux de nouvelles et ainsi de suite). L'adoption par l'industrie des technologies SOA et les normes connexes (comme SOAP ou des services web REST) représente une étape importante pour permettre le partage et l'intégration efficaces des ressources logicielles. La tendance est de créer des services *in the cloud* (hébergées dans les réseau) que les clients peuvent accéder au moyen d'interfaces de programmation mis en œuvre dans des applications de bureau ou des mini-applications installées dans les smartphones (iPhone, les téléphones Android et ainsi de suite).

Grâce à la technologie des services web, les opérateurs de télécommunications peuvent offrir aux développeurs tiers un accès simplifié aux moyens de communication et ainsi les aider à créer des services à valeur ajoutée exploitant leurs capacités réseau.

L'utilisation de plates-formes d'intermédiation (*middleware)* modernes pour mettre en œuvre des services de télécommunications est une étape essentielle pour obtenir un meilleur contrôle sur les coûts de développement et de maintenance. Mais, cela n'est généralement pas suffisant. C'est là où intervient l'ingénierie des modèles. Son rôle est de combler le fossé entre la conception et l'exécution, et plus particulièrement, entre les langages de conception spécifiques à un domaine (VOICE et SPATEL) et les plates-formes d'exécution. Modélisation et génération de code sont des technologies clés pour réaliser le pont entre la conception et l'exécution.

Pour résumer, il ya deux pressions complémentaires qui peuvent potentiellement contribuer de manière significative à accroître l'agilité dans la construction et l'évolution

des services de télécommunication. La première est la «modernisation des plate-formes" - qui est illustré par l'avènement de VoiceXML et les Services Web du SOA. L'autre est le "développement guidé par les modèles (MDA)" - qui, dans notre cas, est illustrée par la définition d'un DSL via la méta-modélisation, et par la création de frameworks d'exécution qui opèrent sur des «modèles» (SPATEL Engine et VoiceBench), ainsi que par le développement de transformations capables de automatiser une grande quantité de l'effort nécessaire pour déployer et tester des services.

Table of contents

1 Chapter - Introduction....1
1.1 Context....1
1.2 Contribution....3
1.3 Outline of the document....4
2 Chapter - State of the Art....5
2.1 Context Overview....5
2.1.1 Model Driven Architecture....5
2.1.2 Service Oriented Architecture....6
2.1.3 Agile Methods....7
2.2 Model Engineering....9
2.2.1 MDA Foundation....9
2.2.2 MDA Standardisation....16
2.3 Service Engineering....22
2.3.1 Specific Vocabulary....22
2.3.2 Integrated composite services and interactive voice services....23
2.3.3 Standards for composite services....23
2.3.4 Standards for Voice Services....29
2.3.5 Standardization of Service Delivery and open APIs....35
2.4 Service Development with MDA....38
2.4.1 Selected research projects....38
2.4.2 Model oriented standards for Services....41
2.5 State of the Art Conclusions....45
2.5.1 Summary....45
2.5.2 Criteria of research....45
3 Chapter - Contribution....46
3.1 Approach for achieving agility in development of telecom services....46
3.1.1 Agility principles for developing telecom services....46
3.1.2 Realizing agility with model-driven technology....49
3.1.3 From the idea of a service to its realization....51
3.2 Composite Services: SPATEL and SPATEL Engine....54
3.2.1 Introduction....54
3.2.2 The SPATEL language....55
3.2.3 The SPATEL Engine framework....68
3.3 Voice-based Services: Voice DSL and Voice Bench....72
3.3.1 Voice DSL....72
3.3.2 Voice Bench Tool Chain....74
3.4 Contribution Discussion....75
3.4.1 MDA Application Issues....75
3.4.2 MDA advantages for service development....80
3.4.3 MDA limitations for service development....82
3.4.4 Summary of contribution....82
4 Chapter - Validation....83
4.1 Validation Overview....83
4.2 Experiments....84
4.2.1 Address book voice service....84

4.2.2 Dinner planning composite service....91
4.2.3 Development of a MDD Tool Chain....97
4.3 Validation Summary....103
5 Chapter - Conclusion and Perspectives....105
5.1 Context of work: MDA and platform modernization....105
5.2 Summary of defended thesis and contribution....106
5.3 Perspectives....107
5.3.1 TelcoML standardization effort....108
5.3.2 Full support for Multi-Modality....108
5.3.3 Model based Natural Language annotations....108
6 Bibliography/References....109
7 Author Publications....119
8 Annex A: Details of Address Book Experiment....i
8.1 Realization with Traditional approach....i
8.1.1 Specification formalism in the traditional approach....i
8.1.2 Implementation of the Address Book service....iii
8.2 Measurements....vi
8.2.1 Measured gain in productivity when using the MDD Voice tool chain....vi
8.2.2 Scope and validity of measurements....vi
8.2.3 Effort needed to specify a functional module....vi
8.2.4 Effort needed to implement a functional module....vii
8.2.5 Corrective factors for design and implementation of a functional module....viii
8.2.6 Effort needed to change a functional module....ix
8.2.7 Productivity measure....ix
9 Annex B: SPATEL Technical Artefacts....xii
9.1 SPATEL metamodel....xii
9.2 SPATEL Textual Grammar....xxi
9.3 SPATEL to WSDL Transformation....xxiv
9.4 Generation of a Service....xxv
9.4.1 The original SPATEL source file in textual format....xxvi
9.4.2 The corresponding SPATEL XMI source file....xxvi
9.4.3 The generated python skeleton code....xxvi
9.4.4 The generated WSDL file....xxvii
10 Annex C: Natural Mashups Experiment....xxix

Figures

Figure 1: Viewpoints of a system....11
Figure 2: Multiple Abstraction Levels....12
Figure 3: Relationships between models and meta-models....13
Figure 4: Transformation definitions with meta-modelling....15
Figure 5 : State Machines in SCXML....34
Figure 6 : Development Process with Apache SCXML....35
Figure 7: SDF Service Component notation....36
Figure 8: Service description according to mTOP MTOSI....37
Figure 9: Participant Specification in SoaML....42
Figure 10 Typical Life-cycle phases....52
Figure 11: Excerpt of SPATEL metamodel....57
Figure 12: Service Interface for a multi-protocol Messaging service....58
Figure 13: Excerpt of State Machine abstract representation....59
Figure 14: Kind of actions....60
Figure 15: Flight Booking Service Example....65
Figure 16: Annotation mechanism in SPATEL....67
Figure 17: Service Creation Process....71
Figure 18: Metamodel for Voice Dialogs....72
Figure 19: Example of voice dialog behavior....73
Figure 20: Architecture of the MDD Tool chain....74
Figure 21: Simulation of a TV Recorder voice interface....75
Figure 22: Vertical Variability....80
Figure 23: Horizontal Variability....81
Figure 24: Main Dialog of Address Book Service....87
Figure 25: Dialog to retrieve contact information....88
Figure 26: Generated code for the Address Book Entity....89
Figure 27: Immediate web execution of the Address Book voice service....90
Figure 28: Dinner planning scenario overview....93
Figure 29: Interface of the Personal Agenda component....94
Figure 30: Logic of the dinner planning service orchestration....95
Figure 31: Activation menu for the dinner planning service....96
Figure 32: Split of activities for tool chain development....98
Figure 33: Lowering the gap between design and implemtation....105
Figure 34: Screenshot of DTMF7 specification....ii
Figure 35: Dialog illustration using parameters....iii
Figure 36: List of property files....iv
Figure 37: State and Transition definition....iv
Figure 38: Implementing decision code....v
Figure 39: Business entity classes for the Address Book....v
Figure 40: Automatic orchestration from natural language request....xxix
Figure 41: Excerpt of natural language configuration for a service....xxx

Figures

Figure 1: Viewpoints of a system....11
Figure 2: Multiple Abstraction Levels....12
Figure 3: Relationships between models and meta-models....13
Figure 4: Transformation definitions with meta-modelling....15
Figure 5 : State Machines in SCXML....34
Figure 6 : Development Process with Apache SCXML....35
Figure 7: SDF Service Component notation....36
Figure 8: Service description according to mTOP MTOSI....37
Figure 9: Participant Specification in SoaML....42
Figure 10 Typical Life-cycle phases....52
Figure 11: Excerpt of SPATEL metamodel....57
Figure 12: Service Interface for a multi-protocol Messaging service....58
Figure 13: Excerpt of State Machine abstract representation....59
Figure 14: Kind of actions....60
Figure 15: Flight Booking Service Example....65
Figure 16: Annotation mechanism in SPATEL....67
Figure 17: Service Creation Process....71
Figure 18: Metamodel for Voice Dialogs....72
Figure 19: Example of voice dialog behavior....73
Figure 20: Architecture of the MDD Tool chain....74
Figure 21: Simulation of a TV Recorder voice interface....75
Figure 22: Vertical Variability....80
Figure 23: Horizontal Variability....81
Figure 24: Main Dialog of Address Book Service....87
Figure 25: Dialog to retrieve contact information....88
Figure 26: Generated code for the Address Book Entity....89
Figure 27: Immediate web execution of the Address Book voice service....90
Figure 28: Dinner planning scenario overview....93
Figure 29: Interface of the Personal Agenda component....94
Figure 30: Logic of the dinner planning service orchestration....95
Figure 31: Activation menu for the dinner planning service....96
Figure 32: Split of activities for tool chain development....98
Figure 33: Lowering the gap between design and implemtation....105
Figure 34: Screenshot of DTMF7 specification....ii
Figure 35: Dialog illustration using parameters....iii
Figure 36: List of property files....iv
Figure 37: State and Transition definition....iv
Figure 38: Implementing decision code....v
Figure 39: Business entity classes for the Address Book....v
Figure 40: Automatic orchestration from natural language request....xxix
Figure 41: Excerpt of natural language configuration for a service....xxx

Abreviations

MDA	Model Driven Architecture
MDE	Model Driven Engineering
SOA	Service Oriented Architecture
IT	Information technology
W3C	World Wide Corsortium (http://www.w3.org)
OASIS	Organization for the Advancement of Structured Information Standards (http://www.oasis-open.org)
OMG	Object Management Group (http://www.omg.org)
SOAP	Simple Object Access Protocol
REST	Representational State Transfer
HTTP	Hyper Text Transfer Protocol
HTML	Hyper Text Markup Language
WSDL	Web Services Description Language
PIM	Platform Independent Model
PSM	Platform Specific Model
AOM	Aspect Oriented Modeling
MOF	Meta Object Facility
XMI	XML Meta Data Interchange
UML	Unified Modeling Language
QVT	Quero, View and Transformations
RAD	Rapad Application Development
XP	Extreme Programming
API	Application Programming Interface
XML	Extensible Markup Language
SDL	Specification Definition Language
SCE	Service Creation Environment

No por mucho madrugar
se despierta el sol más temprano

Acknowledgments

I wish to thank a number of people who have supported, directed, and assisted me in completing this thesis.

First of all I want to thank Bertrand Nicolas which firstly suggested me to initiate this thesis recognizing my expertise in MDA and service engineering.

Secondly I want to thank Jean-Marc Jezequel, my supervisor for his scientific exigency, his time spent on reviewing and providing valuable recommendations.

Third I would like to thank Didier Loustaunau, my manager for his continuous support for completing the thesis despite overwork in my normal activities within my company.

Also I would like to thank various colleagues within France Telecom for the valuable discussions we had, namely Maria Jose Presso, Sebastien Poivre, Jacques Simonin and Gregoire Dupe. The same for others colleagues met in cooperative projects such as J. P. Almeida, Luis Perreira Pires, Paolo Falcarin, Olaf Droegehorn, Marc Born, Olaf Kath, Gilbert Raymond. Also professor Jean Bezivin for introducing me in the very beginning of the French model-driven community (the Groupe Meta).

I also thank all members of the defence committee for their interest in my work.

Finally I would like to thank my wife and children for being at my side.

1 Chapter - Introduction

1.1 Context

Telecom operators are fighting to attract new customers and to fidelize existing customers, by permanently enriching and adapting their service offers. In order to bring such permanent innovation it is necessary to take care of *agility* in the process of service creation.

By agility we should understand at least two main things: firstly, the capacity to put in the market *as fast as possible* innovative services, obviously at reasonable price, and secondly, probably most important, the capacity to achieve the evolution of the existing services so that theiy adapt to the new and often unpredictable expectations from the end users.

The agility in service creation has become to the telco operators a major trend in the competing world of today since it is the richness and accuracy of their offer that will make the difference in respect to the competitors. This is particularly true in a context where new entrants from the IT industry are trying to take advantage of the computing/telecom convergence to dispute to the traditional telecom players the significant revenue of fixed and mobile telephony.

The agility in the telecommunication service creation also impacts third party service providers which are interested to combine key communication functions offered by operators with their own added value functional blocks. On the other hand, for an operator it is important to capture the largest community of third party software developers so that they incorporate in their applications the use of their own network infrastructure (such as a monetized SMS sending component). Co-innovation - partners that agree to work together to share the benefits of an innovation - between operators and 3rd party service providers makes necessary for operators to provide controlled ways to access network resources, which traditionally were strongly locked. This is reflected today by the availability of open APIs published by the operators (for call management, message sending, address book, and so on) like the *Orange Partners* initiative (see http://www.orangepartner.com). Developing APIs to allow developers to access network resources is the first step. From our point of view, the next step to facilitate co-innovation in service development is to agree on the use of high-level modelling formalisms for defining composite services, which are services that aggregate pre-existing building blocks coming either from Telco domain or from IT/internet domain. Part of our contribution was dedicated to the definition of this kind of formalism integrating telecom oriented features.

In this work we will often refer to *models.* A model is *a simplified representation, generally abstract of a process, a system*, intended for describe it, explain it or foresee its behaviour [dict01]. In our specific context, a model is a specification of a

telecommunication service: it describes the offered functions, the structure of manipulated data and its behaviour. It is formalized in the form of a machine-readable data structure to allow further computations and reasoning.

In order to improve time to market the first thing service designers will think is *reusing* already deployed components. Typical design questions are: how can I partition my application for an optimized reuse of modules? What existing components can I use? However, the major risk of reuse is costly integration: sometimes integrating a pre-existing component is worse than re-developing one from scratch. This may arise if the integrated component introduces dependencies that are hard to maintain. In that respect the SOA - Service Oriented Architecture - that focuses on light weight coupling between potentially distributed components, provides an attractive framework to realize agility: the implementation of a service is kept separated from its publication (the API), avoiding difficulties like the constraint to develop components using a unique programming language or the problem of importing incompatible libraries in the same process space. SOA is gaining more and more attention in the telecom industry. The fact that communication facilities like SMS sending and call control are now accessible using web services makes the integration of these facilitates by third party providers near to trivial: no need to be an expert in the telecom domain to use the offered functionalities, if the developer has access to the documentation for accessing and parameterizing the component. Last but not least, in the process of bringing agility in service creation, *standardisation* is expected to play a very important role as a facilitator of integration: the ability to interchange components thanks to some uniformity in the formats and conventions for representing the logic and the data simplifies functional evolution and maintenance of the developed services.

The other aspect of agility in service creation is the capacity of implementing services that can be executed on top of different execution technologies. This applies to the application code running at the server side as well as the application code running at the terminal side (smartphones, television with internet connected and so on) for which there is, nowadays, an incredible heterogeneity of available platforms (Symbian, IPhone, Android, and so on).

In the most general case, a telecommunication service execution implies the launching of one or more parallel threads of logic distributed in various nodes, each one potentially using its own execution technology. Platform heterogeneity occurs here in the context of the invocation of a single service at a given time.

The need for supporting platform heterogeneity also emerges from the need of portability across different terminals that end-users may own: if I want my service to be used by the larger community then I will provide the client software for most popular phone terminals that exists at a given time. Use of *models* combined with various code generators – one for each target platform - is one approach to reduce costs when dealing with heterogeneity. In this case we will attempt to create partial or complete functional service specifications in which we will find no implementation concerns. The SPATEL language we defined as part of our contribution is an example of such a high-level

formalism that simplifies the development of services that are deployable on different platforms.

Multi-platform support may also be motivated by the fact that the technology evolves fast, which may imply recurrent changes in the service execution infrastructure, like for instance changes for solving load balance issues when a service becomes popular.

At this point the techniques of model engineering get on stage: firstly as a conceptual mean for *organizing the separation of concerns* (functional/technical) and then as a *productivity tool* thanks to the automation of a list of coding and testing tasks, which are performed by applying transformations to the original service model.

1.2 Contribution

The thesis defended in this report is that an *appropriate* combination of the two paradigms, the Service Oriented Architecture (SOA) on one hand and the Model Driven Engineering (MDE) on the other hand, is the foundation to offer *agility* in the development process of telecommunication services, that is to say, fast development of new services or evolution of existing ones, taking into consideration typical constraints of telecom operator environments like platform heterogeneity and portability.

More precisely, to gain agility we defend the relevance of an approach based on three points:

- Firstly, the usage of a *high-level executable* formalism for describing services adapted to the inherent complexity of telecommunication services (long running, event based, multi-modality, security concerns, and so on) and aligned with modularity light weight coupling philosophy of SOA. An example of such a dedicated language is the SPATEL language that we have defined and implemented in our experiments. It is based on the state machines execution paradigm.

- Secondly, for an efficient implementation, the usage of a *native* execution framework supporting as directly as possible the high level specification language, serving not only as a simulation environment but also as a "default" execution environment. On top of this framework the designer/developer will be conducted to iterate between the different phases of the service development life-cycle, applying the recipes of fast prototyping on the basis of *executable models*. In our work, the framework implementing natively the SPATEL language is named SPATEL Engine. It offers advanced functionalities to manage the variability in service implementation.

- Thirdly, the development of a series of transformers for addressing the problem of deployment in different target execution platforms and the portability in different kinds of phone terminals. Generic components for model driven development have an important impact as facilitators and accelerators of the software development activity. In the case of our work we have designed and implemented a meta-

modeling framework named PyMOF exploiting dynamic typing on top of Python, and we have provided a significant contribution to the definition of a standardized model-to-model transformation language (QVT/Operational). These two technologies, used separately or in conjunction have helped to develop efficiently the transformers that were needed to build the agile service creation framework.

Last but not least, in this thesis we state the necessity of combining SOA and MDA in an *appropriate* and *pragmatic* way. Despite its potential advantages, exploiting dogmatically model-driven technology may lead to the construction of a service development environment that is too complex to maintain and that at the end provides no agility benefit to service designers.

1.3 Outline of the document

The document first presents a state of the art examining modern practices in model engineering and service engineering, as well as advanced research in combining MDA with SOA. Then the contribution part presents the core of the defended thesis, our vision on coupling SOA and MDA as a way to obtain agility in service creation and evolution. We present the technologies we developed in our work, which are a domain specific language named SPATEL for *integrated composite services* and its associated supporting framework (SPATEL Engine), a domain specific language for interactive voice services named VOICE and its supporting framework (VoiceBench). The last part focuses on the validation of our contribution based on experiments.

2 Chapter - State of the Art

The state of the art is decomposed in five parts. The first part presents an overview of the technological context of our work, mainly a reminder on three major trends of software development: SOA, MDA and Agile Methods. The second part presents and discuss in more detail model engineering, and the third part presents service engineering in SOA context with a panorama of essential industry standards.

The four part focus on combination of MDA and SOA for developing services with focus on development of *composite services* capable of integrating telecom and IT facilities and development of interactive voice based services: what is the state of research and what are the emerging modelling standards in this area.

The last part brings discussions and conclusions on the state of the art and attempts to elaborate some criteria to state the originality and relevance of our contribution.

2.1 Context Overview

2.1.1 Model Driven Architecture

The Model Driven Architecture (MDA) is an approach for developing and maintaining software in which *models* (see definition in Section 1.1) play a central role, as they are directly involved in the production of code. The main objective of MDA is to facilitate *portability*, *inter-operability* and *reuse* (see the MDA Guide [omg-mdag]) through the development of models that realize an appropriate separation of concerns.

The MDA approach promotes the idea that to develop software it is firstly necessary to model the functionalities of the software excluding implementation concerns. Then from the functional model one or more platform dependent implementations are derived by means of transformations, partially or fully automated. Such kind of flexibility may be very important for a company that would need to *migrate* software from an obsolete platform to a modern platform, since the investment made to develop the functionalities is at least preserved. It is also a mean to address portability of software to various platforms following *design once, deploy anywhere* paradigm, which is an evolution of Java's *write once, run anywhere* (WORA) slogan.

In MDA terminology,

- a CIM - computational independent model - represents a description of what the software is expected to do, for instance requirements, domain or business information,
- a PIM - platform independent model - describes the information and computational aspects of software in agnostic way in respect to a family of potential deployment platforms

- a PSM - platform specific model - describes the information and computational aspects of software taking into account platform specificities.

Model transformations take PIM models to create PSMs using additional information - such as marks in the PIM or models describing the target platforms (*platform models*). Then code generation is used to produce the code from the PSM.

Now, in practice there is no obligation to follow this *idealistic* full schema and still be in line with MDA philosophy: one may derive code directly from the PIM without going through an intermediate model. Also, the distinction between CIM and PIM is not always relevant: the PIM used by code generation may include domain and business information.

Two important variations of MDA are:

Aspect oriented modelling (AOM) [Clarke05][Jezequel08]: Various aspects of software functionality - like for instance graphical interface, transactional features and security - are described by separated models. Then these aspects are merged using techniques similar to those used in aspect programming [AspectJ02]. The result of this merging transformation is either an intermediate model or directly the code of the application.

Executable modelling [Harel96][Sunye01][Mellor02]: The PIM contains all the structural and behavioural details needed for an immediate interpretation by an execution engine (or a virtual machine). In that case, there is no need to go into a generation process, except when deploying the software outside the context of the virtual machine. AOM can be used as a mean to make models executable [Muller05].

2.1.2 Service Oriented Architecture

Service Oriented Architecture (SOA) is a paradigm *for organizing and utilizing distributed capabilities* controlled by independent entities [Oasis-rm06]. From an IT perspective, the essential characteristic of an SOA is that it facilitates the integration of software components thanks to *loose coupling*: the producer of a service publish remote interfaces, the consumer of a service can use them without having to know how the service is implemented. The producer may change the implementation without impacting its clients and conversely consumers may change the service provider as far as they found services realizing the same function (with possible adaptation of the interface).

The idea of integrating distributed software based on a well-defined separation between interfaces and implementation and the availability of specific communication protocols for data exchange is not new at all [Andrews00][Mammoud05] (for example CORBA [omg-corba] already realized the vision in 90 decade). But somehow, the web was not as mature as today, in terms of standardization and tooling, to ensure wide adoption by the industry. In the actual technology context, loose coupling is enabled by a list of standards and practices, like SOAP or REST on top of HTTP for communication aspects and WSDL for interface definition. An extensive research work has been conducted to take

advantage of SOA loose coupling principles, especially for *service composition* as a mean to accelerate service development and reuse (see Section 2.2.2 *Selected research projects*). Since the emergence of SOA, many research results have been integrated to the industry through an important standardization effort. In Section 2.3.1 *Relevant Service Engineering standards*, a list of essential standards related to service engineering are discussed.

SOA and Telecom: Service Oriented Architecture (SOA) has emerged as an inescapable paradigm in telecom industry. Adoption of SOA by telecom operators was motivated primarily by the promise that it would facilitate the optimization of the internal information system - thanks to potential re-factoring of functional components. Secondly, it was perceived as a mean to facilitate the exposure of monetizable telecom assets - such as a SMS sending functionality provided as a web service, especially for third party service providers interested to build added-value services exploiting communication facilities. The Telco industry hence become attentive to the elaboration of open standards for enabling the specification of services, as well as interested by frameworks facilitating the implementation and the deployment of such services.

2.1.3 Agile Methods

2.1.3.1 Traditional development of services by telecom operators

The traditional approaches used by telecommunication operators to develop services establish a strong separation between specification and implementation activities. Typically the operator develops a functional specification in natural language complemented with the specification of non functional features (security constraints, performance, and so on). Then a third party company develops the code based on the specification and test cases are defined to validate the developed software. In this scheme, the global architecture of the solution is imposed by the operator but its design is generally out of his control. In some cases, an IT department within the company will be in charge of implementing the service. Nevertheless, the interface between the team in charge of the specification and the team in charge of the development often relies on *manual* exploitation of the specification documents. For large organizations this model of development has the advantage that the responsibilities and expected skills are well established: analysts in charge of the specification do no need to learn about implementation languages, and opposite to this, developers may even need not to know about the requirements of the software. However, the absence of close links between specification and implementation has various drawbacks in terms of agility:

- The operator has little control on the cost for adding new functionalities to the developed services, due to the fact that it has no visibility in the design of the software. As a consequence, even minor changes can have over-estimated and prohibitive costs for the operator.

- The operator cannot easily adjust the desired functionalities due to the lack of lightweight iterations between the specification and the implementation.

- The cost for developing the initial specification might be very high since the operator needs to think on all the details to avoid unwanted interpretations of the specification by the implementer.

2.1.3.2 The Agile Manifesto

Many agile methods have been proposed since 20 years to try to cope with the problems of long-term projects that create products that at the end fail to satisfy customers, not necessarily because of being badly specified, but because the real needs evolved or could not be captured appropriately at the start of the project.

In 2001, a group of experts in software engineering published the Agile Manifesto [Fowler01] formalizing commonalities between various existing agile methods like *Rapid Application Development* (RAD) [Martin91] or *Extreme Programming* (XP) [Beck02]. The emphasis of agile methods is in the continuous involvement of the client in the development of a software product to ensure that it satisfies real needs. This implies a development process that includes iterations and adaptation phases such as the capacity to change the functionalities of the software under development based on received feedback.

The manifesto pointed out four major value statements:

1. The importance of team (interaction of skilled individuals) more than process and tools,
2. Focus on the software to be delivered more than on documentation,
3. Collaboration between clients and developers, more than contract negotiation, and
4. Reactivity to changes requests rather than immutable planning.

It also identifies twelve shared principles that we list below:

P1	"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software"
P2	"Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."
P3	"Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale".
P4	"Business people and developers work together daily throughout the project."
P5	"Build projects around motivated individuals, give them the environment and support they need and trust them to get the job done.
P6	"The most efficient and effective method of conveying information with and within

	a development team is face-to-face conversation.
P7	"Working software is the primary measure of progress."
P8	"Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely"
P9	"Continuous attention to technical excellence and good design enhances agility."
P10	"Simplicity, *the art of maximizing the amount of work not done* is essential"
P11	"The best architectures, requirements and designs emerge from self-organizing teams."
P12	"At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly".

2.2 Model Engineering

A short introduction of MDA was provided in Section 2.1.1. In this section we describe with more details the conceptual foundation of MDA (Section 2.2.1) as well as the key standards to support it, mainly MOF, UML and QVT (Section 2.2.2).

2.2.1 MDA Foundation

This section presents the conceptual foundation of MDA.

Note: The content of this section is a synthesis and an actualization of the MDA assessment made by the partners of the IST MODA-TEL project (http:// www.modatel.org) in Deliverable D2.1 [Belaunde02, Chap1].

2.2.1.1 About Models

The MDA is based on the notion of model. A model is a simplified representation of a system intended for describe it, explain it or foresee its behaviour [dict01]. In MDA context, a model is, more precisely, a representation of (a part of) the function, structure and/or behaviour of a system in a language that has a well-defined syntax, semantics, and possibly rules of analysis, inference, or proof for its constructs [omg-mda]. Examples of models are UML class diagrams [omg-uml], IDL interfaces [omg-corba] and business processes depicted in BPMN [omg-bpmn].

Models can be used in different ways in the course of a development project. When it is used to prescribe properties of a system or system part to be built it is called a *prescriptive* model. When a model is used to describe an existing system or system part, it is called a *descriptive* model.

In the case of prescriptive models, designers produce models of a system introducing information that constrain the intended characteristics of the system being specified. The information required for modelling is obtained along the development trajectory, and documented in several ways.

For several cases, however, a modeller has restricted access to information on the system (part) being modelled. This is often the case for third-party integration, legacy systems and reverse engineering. In these cases, models are described a posteriori, after the system is developed or deployed. Models obtained in such a way are typically black-box models, i.e., models from an external perspective. These models are influenced by the (partial) availability of information on the system being modelled, and may be imprecise because of this.

Another interesting classification is the distinction between *productive* models and *contemplative* models [Bezivin03]: productive models are directly involved in the production of code for a given software whereas contemplative models are used by software designers to share a common understanding of the problem to be solved, before entering in code production. Indeed with the introduction of MDA, models tend to be more productive than contemplative.

In order to understand any non-trivial system, one has to cope with a large amount of interrelated aspects [Jezequel08] [Guizzardi02]. Attempting to capture all aspects of the design in a single model yields too complex and useless models. Therefore, models are derived using specific sets of abstraction criteria, which allow one to focus on particular aspects of the system at a time.

A model is often characterized in terms of the set of abstraction criteria used to determine what is included in the model. Viewpoints, abstraction levels and aspects are examples of abstraction criteria. These concepts are further described in the following sections.

2.2.1.2 Viewpoints

A viewpoint defines a set of related concerns that play a distinctive role in the design of a system. A model defined from a particular viewpoint focuses on the particular concerns defined by the viewpoint. Viewpoints should be chosen with respect to requirements that are the concern of some particular group involved in the design process. The MDA does not prescribe specific viewpoints. Instead, these should be defined by particular design methodologies and the users of such methodologies.

Figure 1 illustrates this notion of viewpoints on a system, with viewpoints V1 to V5. Viewpoints are depicted as spotlights illuminating aspects of concern.

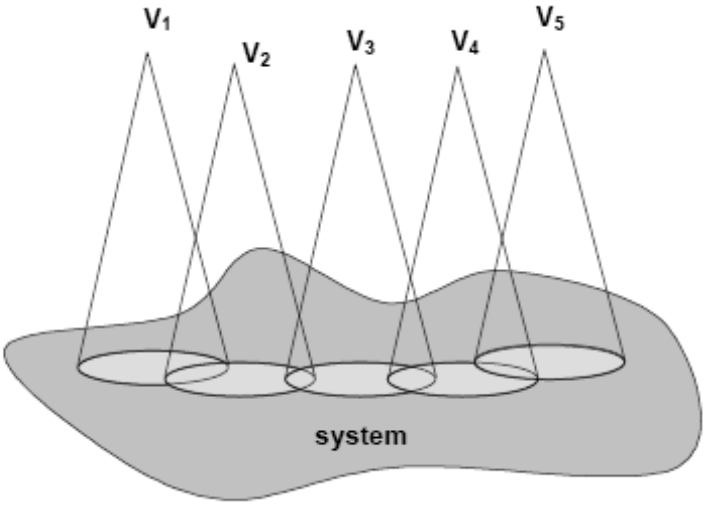


Figure 1: Viewpoints of a system

Examples of viewpoints are the five RM-ODP viewpoints [itu-odp96]:

- The enterprise viewpoint, which is concerned with the business activities of the system being modelled. Examples of models from this viewpoint are business processes described using the BPMN [omg-bpmn]
- The information viewpoint, which is concerned with the information that needs to be stored and processed in the system. An example of model from the information viewpoint is the SID common information model from TeleManagement Forum [tmf-sid10]
- The computational viewpoint, which is concerned with the decomposition of the system into objects that interact at interfaces, and on the constraints on the actions of the objects and the interactions. Examples of models from this viewpoint are WSDL documents [w3c-wsdl07]
- The engineering viewpoint, which is concerned with the mechanisms that support distribution of system parts
- The technology viewpoint, which is concerned with the technological details of the components from which the distributed system is constructed.

The use of different viewpoints in order to describe a system raises the issue of consistency. Descriptions of the same or related entities appear in different viewpoints. Therefore, one must assure that these multiple models are not in conflict with each other. In Figure 1 we denote this by having overlap between viewpoints.

2.2.1.3 Abstraction Levels and Aspects

Abstraction is the process of identifying interesting features of an entity for a specific use [dict02]. In other words, abstraction implies the suppression of irrelevant detail to establish a simplified model. A model M1 is at a higher level of abstraction than a model M2 if M1 suppresses details of the system that are revealed by M2. Specifically, the pair of

models {M1, M2} is in a refinement relationship, in which M1 (the abstraction) is more abstract than M2 (the realization).

Refinement and abstraction are opposite and complementary types of relationships or design activities. Through refinement, an abstraction is made more concrete through the introduction of details, entailing design or implementation decisions, while through abstraction, details of a more concrete abstraction are omitted. In *aspect oriented modelling* (AOM) [Jezequel08] details of high-level abstractions can be partitioned in various concerns and then merged at the level of the concrete abstraction. An important property of either refinement or abstraction is that the resulting abstraction should conform to the original one. In technical terms such conformance can be materialized by the relationship between *classes* in object-oriented programming and *objects* instances of the defined classes. In XML technological space [Kurtev02], the conformance is reflected by the relationship between an XML document and the XML Schema used to validate it. Figure 2 illustrates a number of abstraction levels.

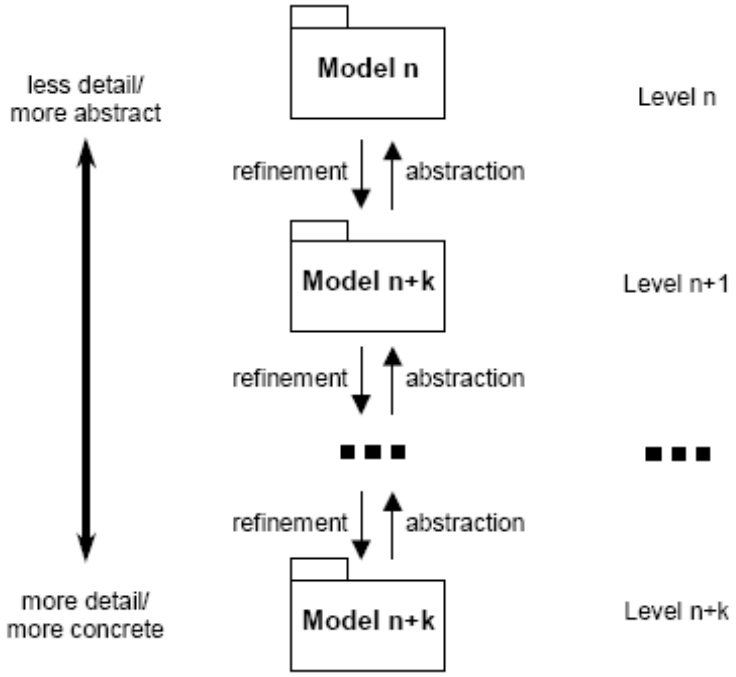


Figure 2: Multiple Abstraction Levels

Design methodologies normally define different abstraction levels to be used for particular viewpoints. In these methodologies, abstraction levels are usually related to milestones in the design trajectory, or are related with particular design goals. Several design methodologies also define refinement (and abstraction) relations in order to guide development of related abstraction levels.

2.2.1.4 Meta-modelling

In the MDA, meta-models are used to support the definition of syntax and semantics of models. When a model B is used to describe the underlying structure of a model A, B is said to be the meta-model of A [Dijkman03][Combemale08]. In an alternative formulation, one can say that the abstract syntax of the model A is defined in the meta-model B [Harel00]. In yet another formulation, one can say that each model element of the model A is an instance of a concept into the meta-model B.

Figure 3 shows an example of a model and its meta-model. The elements of the model are instances of the elements of the meta-model: the classes Employer and Employee are instances of the meta-class Class. The association between Employer and Employee is an instance of the meta-class Association.

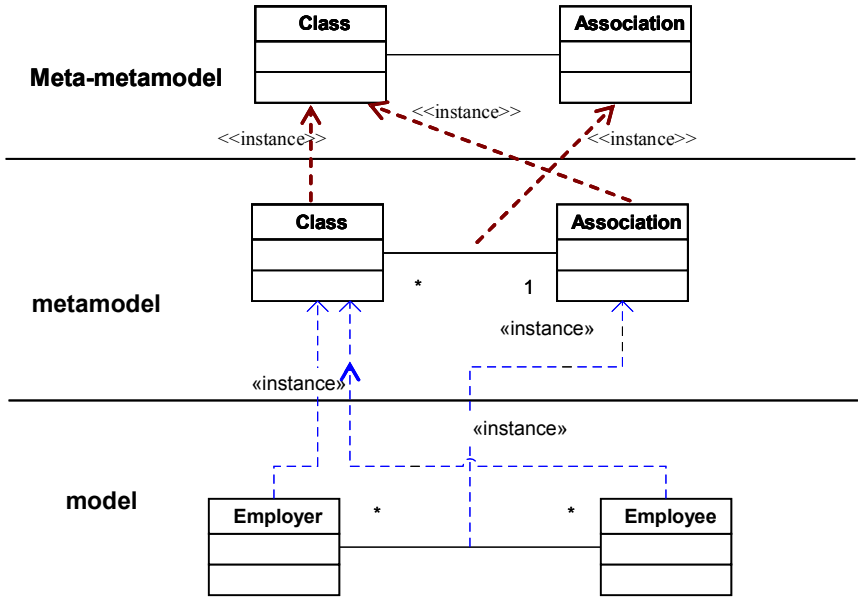


Figure 3: Relationships between models and meta-models

The abstract syntax of a meta-model B can also be described in yet another meta-model C, sometimes called meta-meta-model. This is also depicted by Figure 3. Although the number of meta-levels is arbitrary, meta-modelling frameworks should define a limited number of useful meta-levels. The meta-meta level is often defined as being reflexive, that is to say, with the capacity to describe itself.

An interesting capability of this recursive *instantiation* relationship between meta-levels is that in some cases it can even be used to represent the data level. If the model depicted in Figure 3 (Employer/Employee) can be instantiated (like 'John' and 'Barclays' being instances of Employer and Employee) then the same techniques that are used to

represent models in respect to meta-models can be used to represent in data in respect to the model. This capacity can be exploited in executable modelling environments.

Meta-models are usually accompanied by constraints specifications that restrict the set of valid combinations of model elements in a mode. OMG specifications use a side-effect free expression language named OCL [omg-ocl].

Meta-models are also usually accompanied by natural language descriptions of concepts that correspond to elements of the meta-model, defining informally the semantics of the modelling elements. This approach has been adopted by OMG in the Meta-Object Facility (MOF) [omg-mof] and in the UML proposed standards [omg-uml]. More rigorous approaches define the semantics of modelling elements in terms of a mathematical or formal domain (e.g., the definition of the semantics of the Specification and Description Language (SDL) in [itu-sdl]), or in terms of concrete, formal and explicit representations of domain conceptualisations (e.g., an ontology as proposed in [Sinderen02]).

2.2.1.5 Model Transformation

A particular pattern explored extensively in model-driven engineering is the use of model transformation. Model transformation is basically seen as a mapping of elements of one model onto elements of another model [Sendall03][Mens05]. An instance of usage of this pattern is the creation of software systems by code generation. Each generated artefact, either some code in a programming language or some textual deployment artefact can be manipulated as a model. These models are based on a defined structure, which itself forms a meta-model, which can be expressed in terms of the UML and/or MOF standards.

Transformation is often a refinement in terms of knowledge addition, entailing design decisions. From a broader business-centred model with possible variants or abstract assumptions, a transformation may produce a concrete model in terms of the underlying platform technologies.

Model transformation is useful if formally or systematically defined. As depicted in Figure 4, a transformation may be defined at the level of meta-models. When transformation is applied, a source model is transformed into a target model according to the defined transformation (rules).

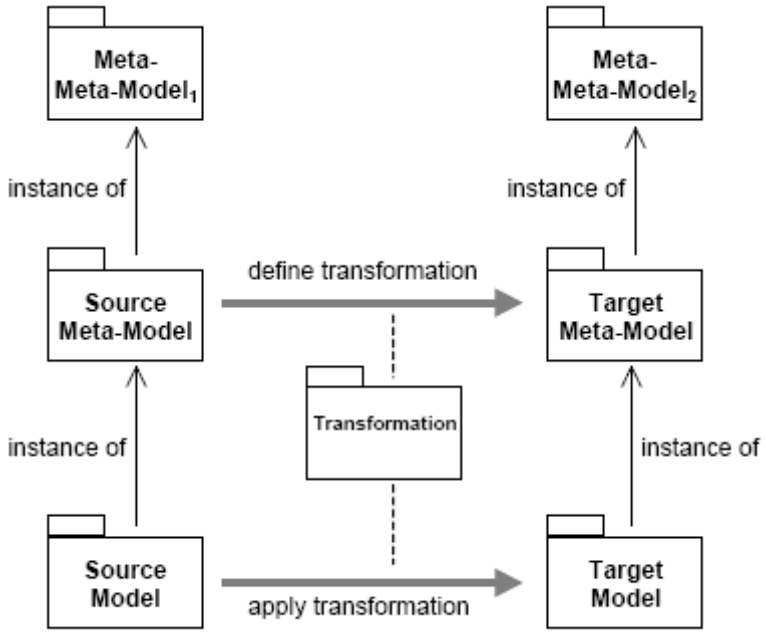


Figure 4: Transformation definitions with meta-modelling

According to OMG definitions, a meta-model is based and constructed from elements of an underlying meta-meta-model (the MOF) and a model is constructed from elements of the meta-model. The use of a common meta-meta-model for the target and source meta-models may facilitate the definition of transformations.

Model transformation can be applied successively. In this case the notions of source and target models are relative. An intermediary model is considered a target model from the perspective of the transformation from the source model, and the same intermediary model is considered a source model from the perspective of the transformation to the final target model.

Transformations can be written using different techniques. The most common way to write transformations in the industry is to use a general purpose language that accesses an API to navigate and create model elements of source and target metamodels (API generation is standardized for various languages including Java). Another approach is to use an executable model-aware language like KerMeta [MullerFleurey05] that hides access to the API. Finally transformation writers may use transformation specific languages such as QVT [omg-qvt] or similar ones like ATL [Bezivin03], VIATRA [Varro02] and UMLX [Willink03], with a large choice in paradigms (graph transformation, pattern-matching, imperative and so on). Among these paradigms, imperative style which makes the steps of the transformation algorithm explicit [Sendall03] is certainly the one that is easier to put in hands of object-oriented programmers, especially when dealing with large scale unidirectional transformations [Patrascoiu04]. The imperative flavour of the QVT standard will be examined and discussed in detail in the next chapter dealing with MDA standards.

2.2.2 MDA Standardisation

The Object Management Group (OMG) has developed a list of standards that provides the basis for developing tools to support MDA approach. Due to their central role in MDA we already cited some of them in this document (that's the case of UML and MOF). In this section we provide a more detailed view on these standards.

2.2.2.1 Meta Object Facility

The MOF (Meta Object Facility) [omg-mof] provides the abstractions for meta-modelling, which in essence specifies how to define models as instantiations of meta-models. Four levels are defined: the meta-meta level represented by the MOF language, the meta-level represented by meta-models defined using the MOF language, the model level, represented by models conformant with metamodels defined in the meta-level. Finally the instance level represents instances of models defined in the model level, which may be simply the data manipulated by programs.

Associated to the MOF specification, the XMI specification [omg-xmi] defines the means for exchanging models using XML. This includes the derivation of an XML schema from a MOF compliant meta-model to validate XML documents representing models. In addition specific mappings define how to derive APIs to manipulate models in general purpose languages (for Java, for Ruby, for CORBA and so on).

In essence MOF concepts are those used in simple UML class-diagrams: we have *classes* and *associations* between classes, owned by *packages*. Classes have *attributes* and *operations* that can be defined locally or inherited from base classes (through the inheritance mechanism). The type hierarchy consists of classes and *datatypes*, which may be structured or be predefined *primitive* types. An annotation mechanism (*tags*) can be used to mark model elements with specific information. Reflection, that is to say the ability to access the meta-level, is provided through special operations on a generic base class named *Object*.

There are two flavors for MOF: one is Complete MOF (CMOF) which includes advanced meta-modeling features like association overriding, and the more basic version named Essential MOF. Actually the most popular industrial implementation of MOF is provided by the Eclipse EMF project [emf] with a variant of EMOF called *Ecore*.

2.2.2.2 Unified Modeling Language

The UML (Unified Modeling Language) [omg-uml] is a general purpose graphical notation for modelling different aspects of software (use cases, scenarios, data, behaviour, deployment and so on). The notation can be particularized to serve the purpose of a specific domain through the *UML profile* mechanism. UML is itself conceptually defined as a MOF compliant meta-model. Conversely, UML class diagrams provides the notation for rendering graphically MOF compliant metamodels.

UML diagrams cover potentially almost all facets of software development. In analysis phase, popular diagrams that are used are use case diagrams and sequence diagrams. In design phase we have a list of structural diagrams like class diagrams and component diagrams and a list of behaviour diagrams like state machines, activities, and collaborations. Deployment diagrams are used to describe deployment of software in distributed nodes.

An important specificity of UML is the customization mechanism called UML Profile.

A UML profile assigns certain elements of the UML meta-model a specific meaning and allows variations of the user interface related to those elements. Main elements involved in a UML profile are:

- Stereotypes of model elements.
- Icons or alternative representations of model elements based on specific stereotypes.
- Tagged Value sets for model elements.
- Constraint for the proper usage of the stereotyped model elements

UML-Profiles are used to visually differentiate model elements. Classes with different stereotypes may be represented with different icons and the definition of tagged values and constraints can be supported by customizable dialogs dedicated to each UML-Profile. Part of the success of UML comes from the ability to use the notation for distinct purposes and in multiple domains [Shani08][Fenster10]

The UML Profile extensibility mechanism play an important role in Domain Specific Languages (DSL) - see Section 2.2.2.4 Domain Modeling and discussion in Section 3.4.1.

2.2.2.3 QVT and Mof2Text

Finally, there is a standard for *specifying* model to model transformations and model to text transformations. QVT [omg-qvt] and Mof2Text [omg-m2t]. These languages provide a *neutral* way to define transformation based on MOF meta-modeling (in the sense that they do not depend on a general purpose language).

QVT has two flavours: QVT Relational is purely declarative and is based on relations and in pattern matching. QVT Operational is imperative. In the following we will concentrate in the later formalism.

Use of imperative logic has some importance in transformation engineering, since it means that the transformation writer will be able to exploit "ordinary" programming skills to solve complex transformation problems and still be able to reason at model level (like organizing its design in terms of meta-model elements to be mapped).

The QVT code below represents an imaginary transformation definition that converts stone to gold. We will use this example to illustrate four noticeable features concerning QVT operational: domain specificity, object-orientation with meta-class extensibility, pseudo-declarative nature and imperative nature. In short the transformation below declares in its signature that it updates models of type MATERIAL, it declares use of two query operations 'getNostradamusFormulaFromStars', and 'isPure' - defined elsewhere, possibly as black-box operations - and defines two mapping rules 'toGold', the first applying generically to all Atoms and the second being specific to Nickel Atoms. More explanations on this example are provided inside the four noticeable features presentation below.

```
transformation StoneToGold (inout model:MATERIAL);
query Atom::getNostradamusFormulaFromStars() : String;
query NickelAtom::isPure() : Boolean;
intermediate property Atom::magicFormula : String;
main() { model->allObjects(Atom)>map toGold(); }

inout mapping Atom::toGold() :Gold {
    self.magicFormula := self.getNostradamusFormulaFromStars();
}
inout mapping NickelAtom::toGold() :Gold  guard {self.isPure();}
       inherits Atom::Gold {
     color := self.electrons>map paintInYellow();
}
```

Domain specificity: QVT/Op is a domain specific language dedicated primarily to model transformation. When looking at the signature of the transformation - StoneToGold in our example - the model practitioner has good chances to immediately understand its goal and the role of its participant models. In line with OCL [omg-ocl], properties and associations defined at meta-model level are directly manipulated without the need of getters and setters operations. In brief, the QVT/Op formalism offers a list of structuring abstractions dedicated to model transformation - like the distinction between query operations - operations that inspect a model to retrieve elements - and mapping operations - that create target elements from source elements. This globally makes a QVT transformation much more readable to transformation practitioners than the equivalent program written in a general-purpose language. QVT code is significantly more compact - a factor of three less than the corresponding JAVA program. In addition, it forces developers to strictly focus on the transformation problem.

Object-orientation and meta-class extension: QVT/Op gives to transformation writers similar mechanisms for reuse and structuring that Java has. A QVT/op transformation behaves like a class: transformation inheritance allows to reuse and to specialize as needed pre-existing transformation definitions to a new context. A specific characteristic of QVT is that mapping operations, query operations and attributes within a transformation can be defined as extensions of the metaclasses involved in the transformation: a simple visitor strategy can then be developed in an elegant way without

forcing a change in the interface of the metaclasses or requiring the definition of complex structures for storing intermediate data. In our "alchemic" transformation example the Atom meta-class is extended with some queries (getNostradamusFormula), mappings (toGold) and new properties (magicFormula).

Pseudo-declarative nature: The QVT/Op language offers various advanced features that somehow "raises the level of expression" compared to ordinary "imperative programming". Two constructs are worth to mention here: guards and fine-grained rule reuse. Guards in mapping rules allows to put some of the decisional logic - selecting the rule to execute in a given context - in the signature of the mapping rather than in the code responsible of rule invocation. By adding such contextual information at signature level, the intent of a rule can more easily be captured. The guard mechanism appears to be particularly powerful when used in conjunction with fine-grained QVT/Op reuse mechanisms like rule inheritance, rule merging and rule disjunction since the actual type of the instance on behave of which a rule is invoked also intervenes in the determination of the rule to invoke. In our transformation example the guard in the NickelAtom::toGuard mapping prevents the magic to apply to non pure elements.

Imperative nature: In QVT/Op, the logic of a transformation is given by a list of rules in which we found explicit sequencing and explicit invocations of other rules. Notice that in our illustrative example the *map* keyword is used to express the invocation of a rule. Explicit sequencing and invocation basically means that almost any transformation that can be written in Java can be reformulated in QVT/op without changing the philosophy and the way of thinking of the original writer. Nevertheless, thanks to the specific constructs presented before, a large QVT/Op definition may really look as a declarative transformation. However, don't be misguided: the ordering in which the QVT statements are written is meaningful: sequencing of instructions and explicit invocation can be exploited intentionally to write in a simple way things that would be, otherwise, in a pure declarative formalism, much more problematic to express. This is especially true in the context of in-place transformations.

To conclude in contrast with most transformation declarative languages which - sometimes provide escape mechanisms for writing specific imperative sections, QVT/Op takes the reverse approach: it is beyond all a uniform imperative language, that looks like a declarative language, but do not requires having to switch between two different ways of thinking to solve specific transformation problems.

2.2.2.4 Domain Modeling: MOF and UML Profiles

To create domain specific languages (DSL), two approaches exploiting MDA standards can be used: MOF metamodeling or UML Profiles. In the case of MOF, the DSL is defined by a MOF metamodel, which may be created from scratch or be defined as an extension of an existing metamodel (by means of package import construct). In the case of UML Profile, by construct the DSL is defined as a specialization of the UML metamodel.

An important debate has traversed the last decade the modeling community to know which of the two approaches is to be preferred. In [Desfray00] the author emphasizes *flexibility* in changes as the main differentiator of UML profile technique, which leads to the conclusion that metamodel technique is appropriate when domain concepts are stable and standalone whereas profile are to be preferred when domain concepts are subject to frequent changes and subject to combinations with other domain models.

Other authors (like [Brockmans06]) focus on the distinction between abstract syntax and concrete syntax and envisages complementary usage of both techniques. In Section 3.4.2 *MDA Application Issues*, we elaborate our point of view on this question, which is important since it impacts significantly the architecture of a service creation environment (SCE) that would be build with MDA.

Beyond the problem of "meta-modelling versus UML profiles" we have at least two distinct questions:

- What is the better technique to define the abstract syntax of a specific domain?
- What graphical concrete notation should I use for my domain? Can I use the UML diagrams for this purpose?

  The ambiguity regarding UML Profiles is that the motivation for using it may be reuse of UML abstract syntax or reuse of UML graphical notation or a mix between the two.

Defining the abstract syntax

For the former question, there are distinct approaches that are currently being used.

- A meta-model is defined completely from "scratch". It's often the case for "domain" meta-models that are not too larger. For the complex parts of the meta-model it often a good practice to "copy/paste" patterns from other meta-models [Kobryn00]. As an illustration of this, the behaviour part of the SPATEL metamodel presented in Figure 11 in Section 3.2.2.2 was partially copy/pasted from the UML metamodel.
- A standalone meta-model can be defined as an extension of an existing meta-model, using the standard MOF extensibility mechanism (Package import). This is for instance the case for specialized usages of CWM [omg-cwm]. In some sense the SPEM meta-model [omg-spem] is an example of this case since the "foundation" package is an excerpt of the UML meta-model.
- A meta-model can be specialized using any usable ad-hoc extensibility mechanisms (typically through annotations, if the metamodel has the *annotation* concept). In UML context, light-weight extensibility is provided by the UML Profile mechanism. Since UML 2.0, the equivalence between a UML Profile and a MOF compliant metamodel has been formally defined,

where a Stereotype becomes a metaclass (see in [omg-umlinfra], Section 13.1.2 Extension).

At first sight, the third option, having an equivalence between a UML Profile and a MOF metamodel appears as reconciling the two approaches. But in reality the important point, which is not always well understood, is that, despite the fact that for each UML profile there is a corresponding MOF meta-model, in most cases, the "equivalent" MOF meta-model is far from reflecting a "clean" semantic description of a domain. Because the leading motivation for using UML is the graphical notational support, the equivalent MOF meta-model resulting from a UML profile definition, will in practice contain a lot of redundancies and a lot of unnecessary complexity. The XMI rendering and the operational APIs resulting from this "equivalent" meta-model would be too complex in comparison with the ones that could be obtained from a direct model of the domain (the standalone "domain" meta-model).

An example of this "pollution" was the definition of SPEM 1.0 metamodel [omg-spem] which was built as an extension of the UML 1.4 metamodel. The advantage was direct reuse of UML activity diagrams - no need to re-invent the wheel to express logic - but the disadvantage was redundancies in the representation of tasks (use case notational view versus activity notational view).

As a conclusion, for the former question, we will say that even if it is possible to make proper domain meta-modelling with UML profiles, in practice this is not easy because it's very difficult not to be influenced by the notational aspects which tend to pollute the meta-model.

For the second question, which deals which the concrete notation and tool selection, traditionally each domain has its specific set of modelling tools dedicated to the domain. Currently the UML based tools and the meta-case tools are positioning as competitors of this tools. These domain specific modelling tools are often very expensive and lack support of the model-oriented import and export standardized facilities (such as XMI support). In the other hand they are known as being more mature for the domain perspective.

A meta-case tool approach, in which the concrete graphical notation can be build from scratch and attached to each meta-class, is very attractive. Some existing tools provide such facilities - for instance MetaEdit+ [metaedit].

However, the usage of UML based notations for domain specific purposes have also a lot of advantages:

- Availability of the UML tools at relatively low costs
- Reuse of stable and standardised notation which means less cost to learn and understand it
- Profile support in some UML tools which allows a high degree of customisation (the tool behaves mostly as a meta case tool).

In Section 3.4.1 MDA Application Issues, we provide our point of view on this debate.

From a theoretical point of view, both approaches are in fact complementary and play different role [Brockmans06]: MOF defines the abstract syntax whereas a UML Profile may provide a specific concrete syntax (see Discussion in Section 2.3). Moreover, a bi-directional mapping can be defined to link selected UML concepts to those in the metamodel. Now, in practice, maintaining such complementary representation may be costly.

2.3 Service Engineering

In this section we describe with some detail service engineering practices - mainly related to the service oriented architecture (SOA) - but independently of the introduction of MDA. Examination of research regarding the combination of SOA and MDA will be provided in Section 2.4.

By service engineering we mean in fact the techniques used to describe services and to implement services in compliance with the service description. Our study will be focused on two kinds of services: integrated composite services and interactive voice services.

2.3.1 Specific Vocabulary

We provide here some clarification in the vocabulary we will use concerning service engineering in this section and in the rest of the document.

Service versus application

From an IT perspective, a service is a *mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface* [oasis-rm06]. An application is software *providing a set of specific functions to users dedicated to a business task* [dict03]. An application may exploit various services. Some applications have as unique purpose to provide user-friendly access to the functionalities of a service. This is often the case for mini-applications found in modern smartphones stores (like the Android Market, and the Apple Store)

Server side versus terminal side service deployment

The implementation code of a service can be deployed in application servers owned by the service provider but some parts may be deployed directly in the terminal of the user. The latter case does not only concern GUI aspects but also the manipulation of local resources (like the GPS module for geo-localization). .

Service specification versus service design

A service specification describes functional and non functional properties of the service, in principle, independently of any implementation issues. Typically a service specification contains information on how the service behaves (useful to implementers of the service) and information how to access it (useful to implementers and to third party developers).

A service design describes how a service is implemented. It typically describes how the software is organized (architecture) and includes abstract representations of the software to be implemented - like class diagrams reflecting future code.

2.3.2 Integrated composite services and interactive voice services

An integrated telecom service is a service that exploits the convergence of communication networks - landline, wireless and voice, and in the same time takes advantage of facilities accessible from the WEB. The SOA plays an essential role for the development of integrated services because it simplifies significantly the integration of different kinds of technology [Baravaglio05]. For instance, to take advantage of SMS capability, a third party developer that can use remote APIs provided by the operator will not need to be expert in telecoms to be able to integrate the capability in its application. From a programming point of view invoking a remote service behaves as a plug-and-play functionality: no need to install and re-compile external software.

Voice applications are software applications that allow people to interact with a machine using voice. The machine in question is what is called an Interactive Voice-response Server (IVR). Because dialog interaction is generally complex, developing voice applications typically involves the usage of a dedicated language defining the interaction between the human and the machine.

2.3.3 Standards for composite services

The W3C and OASIS standardization bodies have defined several standards related to web services. In this chapter we focus on major ones related to service definition: WSDL, SA-WSDL and BPEL. When possible we mention criticism on these formalisms and their relevance in respect to the telecom industry.

2.3.3.1 Service Interface definition (WSDL 2.0)

Overview of WSDL

The Web Services Description Language (WSDL) in its latest 2.0 version is a W3C recommendation since 1997 [w3c-wsdl]. It has been promoted by two major actors in IT

industry Microsoft and IBM. The purpose of WSDL is to allow the declaration of service interfaces accessible through the web. Interfaces are described firstly in an abstract way (independently of communication protocols and implementations) to promote reusability. It also contains a concrete section indicating the protocols used and the access points. In short in WSDL a service is described in the following way: an interface contains a collection of operations, each operation declares inputs and outputs parameter, as well as input and output faults. Each parameter has a type which is provided in the form of a XML Schema (embedded or referenced in the WSDL document). An operation declares the message-exchange pattern (like request-response) to be used. After that the WSDL file may include one or more bindings indicating the protocols used (such as HTTP or SOAP) and specific configuration parameters (like input serialization encoding). Finally the end-points for each binding are provided.

An example of a service interface declaration is provided below (this is taken from http://www.w3.org/TR/2007/NOTE-sawsdl-guide-20070828/):

```
<wsdl:description
 targetNamespace="http://org1.example.com/wsdl/CheckAvailabilityRequestService/"
  xmlns="http://org1.example.com/wsdl/CheckAvailabilityRequestService/"
  xmlns:wsdl="http://www.w3.org/ns/wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <wsdl:types>
   <xsd:schema targetNamespace="http://org1.example.com/wsdl/CheckAvailabilityRequestService">
    <xsd:element name="CheckAvailabilityRequestServiceRequest">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="itemCode" type="xsd:string"/>
          <xsd:element name="date" type="xsd:string"/>
          <xsd:element name="qty" type="xsd:float"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="CheckAvailabilityRequestServiceResponse" type="itemConfirmation"/>
    <xsd:simpleType name="itemConfirmation">
      <xsd:restriction base="xsd:boolean"/>
    </xsd:simpleType>
    </xsd:schema>
  </wsdl:types>

  <wsdl:interface name="CheckAvailabilityRequestService">
   <wsdl:operation name="CheckAvailabilityRequestOperation" pattern="http://www.w3.org/ns/wsdl/in-out">
     <wsdl:input element="CheckAvailabilityRequestServiceRequest"/>
     <wsdl:output element="CheckAvailabilityRequestServiceResponse"/>
   </wsdl:operation>
 </wsdl:interface>
</wsdl:description>
```

Discussion on WSDL

WSDL is nowadays well supported in almost all existing web service frameworks and is already used by telecom operators to publish open APIs of telecom facilities (SMS sending, localization and so on).

However there is some criticism on WSDL formalism. One major problem is the complexity of the type system - based on XML Schemas - used by WSDL to define the structure of the exchanged data [Martens05]. Because WSDL offer too many alternatives to

structure the information related to service parameters, in practice most web-service frameworks (like AXIS [Volkmann02]) impose their own conventions which differ from the conventions chosen by other frameworks. As a consequence tool interoperability is not optimal: we cannot easily reuse WSDL definitions produced with one tool in another tool. The worse situation indeed comes when WSDL files are edited manually since we cannot guaranty any homogeneity in the organization of data.

Another problem concern the mixing of abstract definition - like operation signatures - and concrete information (explicit bindings and URLs for endpoints). The fact that in a WSDL file there is an abstract definition part (interface and operations) then a concrete definition part (bindings and endpoints) has the advantage that all the information needed to operate with the service is put in a single place. However, this has the drawback of mixing information of different nature and has the effect of making the definition verbose - compared to simple usage of textual or graphical language like CORBA IDL or UML. In practice we observe that the concrete part of a WSDL file is not used in execution tools; original design-time local endpoints such as "http://localhost/somethingelse" need anyway to be replaced to access the deployed web service.

So to summarize, WSDL is of major importance in telecom to expose web service interfaces - because it has the invaluable characteristic of being already a well accepted and supported standard. However for maintenance reasons, it is preferable to adopt a process in which WSDL documents are generated, rather than being written manually.

2.3.3.2 Service Interface definition with semantics (SA-WSDL)

Overview of SA-WSDL

The *Semantic Annotations for WSDL* (SA-WSDL) is also a W3C recommendation adopted by W3C since August 2007 [w3c-sawsdl]. It allows adding semantic annotations to WSDL elements, such as categorization information to facilitate the publishing of the service and service discovery. Two semantics annotation constructs are defined by the specification: one is the attribute *modelReference* to link an element in the WSDL description to an element in a semantic model (for instance an OWL class) and the other are two attributes named *liftingSchemaMapping* and *loweringSchemaMapping* which are used to indicate syntax mappings between the referenced semantic data and the actual type in the WSDL document.

Below we provide an example of an annotation intended to categorize a service interface (this example is also taken from the SA-WSDL User Guide). The annotation references here an ontology elements defined separately using RDF format [w3c-rdf] (not shown here).

```
...
<wsdl:interface name="CheckItemAvailabilityRequestService"
  sawsdl:modelReference="http://www.w3.org/2002/ws/sawsdl
```

```
          /spec/examples/taxonomy
          /POServiceClassification#ItemAvailabilityCheck">
      ...
</wsdl:interface>
    ...
```

Discussion on SA-WSDL

SA-WSDL is potentially an important standard for telecom industry. It is an attempt to make web services descriptions ready for the emergence of the so-called *semantic web* [Berners01] - where services expose intelligent information about themselves to facilitate automatic reasoning (useful for instance for dynamic service composition).

A clear advantage of the approach taken by SA-WSDL, which is based on annotations linking WSDL elements to external elements defined by some ontology, is that there is no need to create a new formalism for semantic definition dedicated to service definition. In fact we can link any existing suitable formalism for describing semantics. Now, as pointed out in [Chabeb08], an important problem we found in SA-WSDL is that the annotation placed in WSDL elements is too minimalist, since not dedicated to describe behavior: modelReference attribute is used for all kinds of semantic information we would like to refer. Hence we cannot infer from the reading of the annotated WSDL file the exact meaning of the added links: for instance, is a *modelReference* attribute placed in an interface element intended to indicate the "goal", a "preconfition", an "effect" of the service or it is merely intended to classify it within a category of services?). Also, due to this restriction, the annotation mechanism of SA-WSDL cannot be used as such for other similar needs that are important in telecom field, like QoS features and more generally non functional features.

Apart this consideration, SA-WSDL standard relies on WSDL and as such has all the advantages and caveats of it (see WSDL discussion in Section 2.3.2.1). In particular it mixes abstract information on the service with more implementation-dependent information like protocols used and hard-coded end-points.

The SA-WSDL standard does not make assumptions on how the annotations are added and managed. If we follow a generative approach for WSDL files, exploiting a high-level service description formalism, clearly SA-WSDL files also would be generated, assuming there is a equivalent formalism for attaching semantic data in the high-level service description. The SPATEL language we describe in the *Contribution* in Section 3 of our thesis document exposes such a feature.

2.3.3.3 Service Orchestration (BPEL)

Overview of BPEL

The BPEL 2.0 language has been standardized by OASIS consortium since 2007 [oasis-bpel]. The purpose of this specification is to allow specifying abstract or executable processes, typically involving the invocation of more than one web service. An executable

process represents an internal view, whereas an abstract process represents an external view which is intentionally left incomplete. Because the intelligence of the execution is specified in a centralized way, a BPEL process represents an orchestration of services - which is traditionally distinguished from service choreography, where the logic of execution is distributed between the participants.

In BPEL the process logic uses structural programming constructs to deal with conditional execution (if-then-else), loops, sequencing or parallelization of commands. Detailed computations (what is commonly called as "programming in the small" contrasted with "programming in the large") can be done using XPATH [w3c-xpath] or any other expression language. For instance, BPELJ language variant [Blow04] allows embedding Java code to specify data computations. An important characteristic of BPEL is support of transactional features which are typical to long-running processes. For instance compensations actions can be defined in case of failures.

In BPEL the situation where a process is interrupted waiting for an event to occur is represented by the *receive* construct. This makes BPEL usable to describe long running processes - which is a characteristic of the majority of business processes involving people. We should note that a variant of BPEL named BPEL4People have been proposed to specifically support process with human intervention [oasis-bpel4p].

The BPEL language assumes reuse of WSDL documents when referring to the invoked services. It does not define a graphical notation.

For illustration purpose we provide below an excerpt of the BPEL specification of the very famous "purchase order" process taken from the BPEL specification. In this example we see the usage of "sequence" and "flow" controls as well as an explicit invocation of a web service (using the "invoke" construct) that refers to a WSDL abstract service operation "requestShipping" (no reference to concrete bindings to optimize reuse).

```
<sequence>
   <receive partnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="PO"
      createInstance="yes">
      <documentation>Receive Purchase Order</documentation>
   </receive>

   <flow>
      <documentation>
         A parallel flow to handle shipping, invoicing and
         scheduling
      </documentation>
      <links>
         <link name="ship-to-invoice" />
         <link name="ship-to-scheduling" />
      </links>
      <sequence>
         <assign>
            <copy>
               <from>$PO.customerInfo</from>
               <to>$shippingRequest.customerInfo</to>
            </copy>
         </assign>
         <invoke partnerLink="shipping" portType="lns:shippingPT"
            operation="requestShipping"
            inputVariable="shippingRequest"
            outputVariable="shippingInfo">
            <documentation>Decide On Shipper</documentation>
            <sources>
               <source linkName="ship-to-invoice" />
            </sources>
         </invoke>
```

Discussion on BPEL

BPEL

The BPEL standard has been designed to *orchestrate* web services and to that end it contains all the ingredients we could expect to achieve such kind of task. Saying that it is a business process execution language is however a bit abusive in the sense that not all business processes can accurately be modelled as an orchestration [Korp02] [Vigneras08]. For instance, in our day life many tasks are essentially incremental and permanently active with no evident predecessor or successor.

In terms of execution model BPEL is closer to activity diagrams style found in UML rather than state machines, despite the fact that the *receive* construct in BPEL involves conceptually a waiting state. By the way, an alternative mean for expressing complex orchestrations is the usage of plain state-machine based formalisms, such as the SXCML presented in this document in Section 2.3.3.3. But, in the other hand SCXML, not

being specialized to business process, does not contain some first-class interesting features we found in BPEL (like compensation).

The fact that BPEL has poor support of local computations ("programming in the small"), as pointed in [Blow02] to motivate BPELJ variant also constitutes a barrier for using it. Local computations can be needed in between two service invocations to realize data conversions. At the end, some organizations tend to prefer using general purpose languages to implement service orchestrations. The advantage is that they do not have to deal with a specific runtime for executing their assemblies and they are not constrained by the verbosity of the formalism. Disadvantage is that the service logic is not anymore exposed in a *clean* way and is becomes not agnostic in respect to the programming language.

Another problem is that BPEL foundation is totally tied to the web service technology. However, in real life services can be of different nature, especially in telecom, not necessary exposed as remote web services. Accessing to a "geo-localization" facility in a smartphone environment may for instance imply accessing a local resource in the phone rather than a web service provided by the operator. Making this transparent in a BPEL file implies hence to perform a significant work for encapsulating these local services so that they become visible to the BPEL engine as ordinary web services.

2.3.4 Standards for Voice Services

The standards for the development of voice service are from the World Wide Web consortium (W3C). In this section we describe those that are relevant to our study.

2.3.4.1 VoiceXML

Overview of Voice XML

VoiceXML [w3c-voicexml] (also known as VXML) is a specification of the World Wide Web Consortium (W3C). It provides means for specifying the interaction between humans and a machine exploiting voice recognition and voice synthesis. With VoiceXML, voice applications can be developed and deployed in a similar way than ordinary web applications using HTML language and JavaScript. VoiceXML documents are interpreted by a voice browser, which is generally connected to the telephony network of an operator. VoiceXML pages can be generated dynamically by a HTTP server hosting the logic of the voice application.

The actual version of VoiceXML is 2.1. A working draft of version 3.0 is available since December 2009.

To illustrate a simple usage of VoiceXML, we provide here the specification of voice interaction for a coffee machine (the example is taken from VoiceXML 2.0

specification). In this example the machine asks the user for a choice of drink and then submits it to a server script:

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
  <form>
  <field name="drink">
     <prompt>Would you like coffee, tea, milk, or nothing?</prompt>
     <grammar src="drink.grxml" type="application/srgs+xml"/>
  </field>
  <block>
     <submit next="http://www.drink.example.com/drink2.asp"/>
  </block>
 </form>
</vxml>
```

Below an example of interaction corresponding to the VXML source above:

C (computer): Would you like coffee, tea, milk, or nothing?
H (human): Orange juice.
C: I did not understand what you said. (a platform-specific default message.)
C: Would you like coffee, tea, milk, or nothing?
H: Tea
C: (continues in document drink2.asp)

The essential design concepts in VoiceXML are presented briefly below:

A *document* (or a set of related documents called an *application*) forms a conversational finite state machine. The user is always in one conversational state, or *dialog*, at a time. Each dialog determines the next dialog to transition Transitions are specified using URIs, which define the next document and dialog to use. There are two kinds of dialogs: *forms* and *menus*. Forms define an interaction that collects values for a set of form item variables. A *subdialog* is like a function call, in that it provides a mechanism for invoking a new interaction, and returning to the original form. Variable instances, grammars, and state information are saved and are available upon returning to the calling document.

Discussion on Voice XML

The VoiceXML executable language standardized by W3C is the reference in the industry for implementing voice interactive services. Nowadays we can hardly imagine developing an IVR without having this technology being used at some point in the execution of the service, taking into account the wide availability of supporting tools and its maturity.

However VoiceXML remains a language appropriate for experimented developers; it is not intended for service designers that have no programming background. Also to take into account data changes at runtime (like contact persons in an address book), in most voice large-scale voice interactive applications, VoiceXML pages are not manually edited but are generated on demand to execute specific portions of the application logic (in contrast with the option to have a large file containing all the logic). This means that VoiceXML is not necessarily the language to be directly used by developers. The preferred approach, from our experience, is to use a high-level language describing the complete service logic of the application. Such high-level definition is then used at runtime to generate on demand the VoiceXML pages representing the portions of the application logic to be executed.

2.3.4.2 CCXML

Overview of CCXML

The Call Control Extensible Markup Language [w3c-ccxml] is another specification of the World Wide Web Consortium (W3C). It provides means to express usage of call control functions such as establishing a call, hang up, transfer and so on. It can be used in combination with VoiceXML by enhancing or replacing call control constructs that exist in VoiceXML. Actual version is V1.0 published in January 2007.

The example below, taken from CCXML v1.0 specification, shows how the connection with VoiceXML can be done. The application answers an incoming phone call and then connects it to a VoiceXML dialog that returns a value that is then logged to the platform:

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0" xmlns="http://www.w3.org/2002/09/ccxml">
  <!-- Lets declare our state var -->
  <var name="state0" expr="'init'"/>

  <eventprocessor statevariable="state0">
    <!-- Process the incoming call -->
    <transition state="init" event="connection.alerting">
      <accept/>
    </transition>
    <!-- Call has been answered -->
    <transition state="init" event="connection.connected">
      <log expr="'Houston, we have liftoff.'"/>
```

```
      <dialogstart src="'dialog.vxml'"/>
      <assign name="state0" expr="'dialogActive'" />
    </transition>
    <!-- Process the incoming call -->
    <transition state="dialogActive" event="dialog.exit">
      <log expr="'Houston, the dialog returned ['
              + event$.values.input + ']'" />
      <exit />
    </transition>
    <!-- Caller hung up. Lets just go on and end the session -->
    <transition event="connection.disconnected">
      <exit/>
    </transition>
    <!-- Something went wrong. Lets log some info and end the call -->
    <transition event="error.*" >
      <log expr="'Houston, we have a problem: (' + event$.reason +
')'"/>
      <exit/>
    </transition>
  </eventprocessor>
</ccxml>
```

The invoked VoiceXML file will be:

```
<?xml version="1.0"?>
<vxml xmlns="http://www.w3.org/2001/vxml" version="2.0">
  <form id="Form">
    <field name="input" type="digits">
      <prompt>
        Please say some numbers ...
      </prompt>
      <filled>
        <exit namelist="input"/>
      </filled>
    </field>
  </form>
</vxml>
```

As depicted by the examples, we can see that CCXML defines its logic flow in the form of a transition-oriented state machine (focus on what happens in transitions rather than in states). It has specific constructs for treating calls, but apart of this is has the typical constructs for defining complex flows (if/else, loops).

Discussion on CCXML

CCXML being a specific call control language has various constructs to deal with call management (createcall, createconference and so on). Now, from the point of view of voice application design, we do not found that CCXML brings a significant added value,

since most of the primitive telephony functions are already integrated to VoiceXML. The specific ones (like conferencing) not existing in VoiceXML could be supported as invocations to external entities, typically through the form of web services accessible by HTTP calls. This has the advantage to be applicable to other specific facilities not natively supported by CCXML (like presence and localisation).

Nevertheless the CCXML language can be used in other contexts than voice application development. For instance a CCXML interpreter capable of executing CCXML documents can be used as the core technology to implement individual telephony web services (stateless services) or conversational ones (statefull). The implementation of stateless services in that case is quite simple since it consists of small pieces of SCXML.

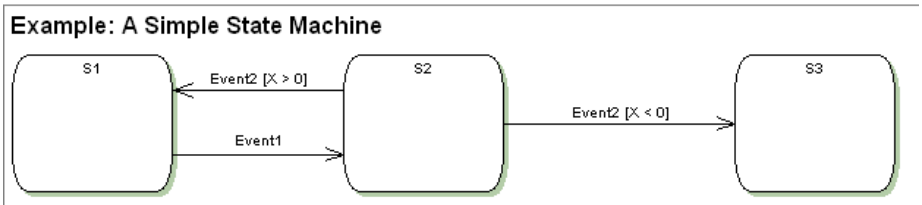
2.3.4.3 SCXML

Overview of SCXML

State Chart XML (SCXML) [w3c-scxml] is a specification of the World Wide Web Consortium (W3C). It provides a generic state-machine execution environment based on Harel State Tables [Harel87]. It also defines mappings with UML state machines to allow using UML as a graphical syntax.

SCXML, although not specifically designed for voice applications, can be used to develop voice applications. To that end SCXML specification can reference VoiceXML XML documents.

Here's an example which shows how a simple state machine - depicted in UML - is encoded in SCXML.



SCXML Equivalent

```
 <?xml version="1.0" encoding="us-ascii"?>
<scxml version="1.0" xmlns="http://www.w3.org/2005/07/scxml">
  <state id="S1">
    <transition event="Event1">
      <target next="S2"/>
    </transition>
  </state>

  <state id="S2">
    <transition event="Event2">
      <target next="S1"
    cond="X&gt;0"/>
    </transition>
    <transition event="Event2">
      <target next="S3"
      cond ="X&lt;0"/>
    </transition>
  </state>

  <state id="S3">
  </state>

</scxml>
```

Figure 5 : State Machines in SCXML

SCXML can represent arbitrary complex state machines, with sub-states and parallelism. It can be seen as a simplified XML representation of UML State Machines (version 1.5, not UML 2 which has introduced some more concepts).

Commons SCXML [cscxml] is an open source implementation of the SCXML W3C standard. It is mainly composed of a Java SCXML engine capable of executing a state machine defined using a SCXML document, while abstracting out the environment interfaces.

Figure 6 (taken from site: http://commons.apache.org/scxml/) shows the normal process for creating and executing SCXML applications with the Apache implementation:

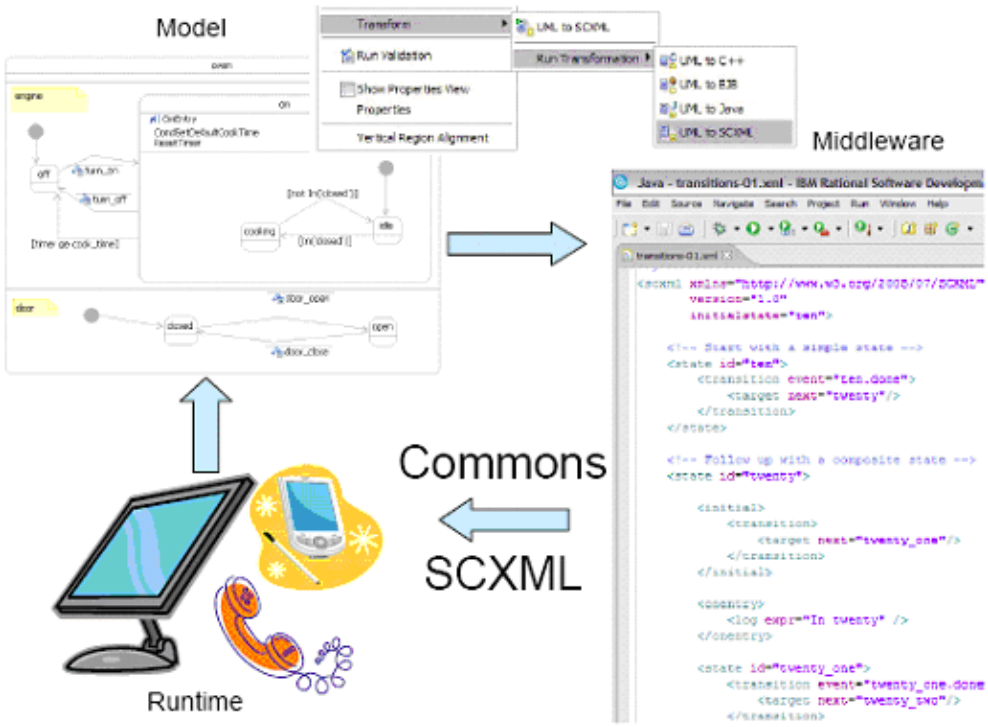


Figure 6 : Development Process with Apache SCXML

Figure 6 shows an example of UML state machine re-interpreted as a SCXML definition which is then deployed in the Commons SCXML environment so that in can be executed in a real environment.

Discussion on SCXML

The SCXML, as a generic formalism for expressing state machines, can definitely play the role of a control language for VoiceXML. Rather than using the dialog transition mechanism inherent to VoiceXML one may coordinate the sequencing of VoiceXML pieces using the SCXML transitions and other control flow primitives. Beyond its usage in voice services, SCXML can be used also to formalize service compositions, especially for statefull and long-running services.

Anyway, SCXML like VoiceXML remain a programming language rather than a design language. The mapping with UML state machines is, by the way, not provided formally in the W3C specification, but only illustrative samples of the mapping are given.

2.3.5 Standardization of Service Delivery and open APIs

In this section we examine two important telecom-specific standardization initiatives which are relevant to the topic of our study: the first, coming from the Telecom Management Forum (TMF), named Service Delivery Framework (SDF), aims at defining a

common understanding of all functions required for the lifecycle of a service delivered to a customer, and the second, coming from the Open Mobile Alliance (OMA) aims at defining concrete APIs to simplify the access by third parties to telecom resources.

2.3.5.1 TMF Service Delivery Framework initiative

Overview of SDF

The Service Delivery Framework (SDF) unifies under a logical view various aspects of service delivering: design, deployment, activation, provisioning, sale, execution, charging, billing, retirement, trouble resolution and so on.

SDF manages three main artefacts with their own lifecycle: Product, Service and Resource. A Product is bundling of Services and Physical Resources (equipment model) to make available to Customer. A Service requires Logical Resources (capabilities) which are provided by Physical Resources. TMF reuses TMF NGOSS concepts [tmf-etom] [tmf-sid] for service and resource management: sTOM (process view), SID (Information view), TAM (application view).

SDF uses a notation similar to UML composite component to describe a service component, as depicted by the figure below:

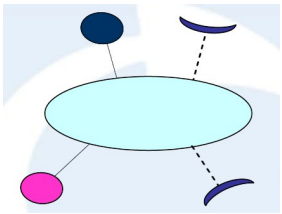


Figure 7: SDF Service Component notation

In this notation three kinds of interfaces are represented: a service functional interface (blue ellipse in the top), a service life-cycle interface (pink ellipse in the bottom) and one or more service interface customer. In parallel to this a conceptual model of service description structure called mTOP MTOSI has been defined. As depicted by Figure 8, a service interface includes an information model, one or more behavioural models (composition, choreography, business operations) and non functional requirements.

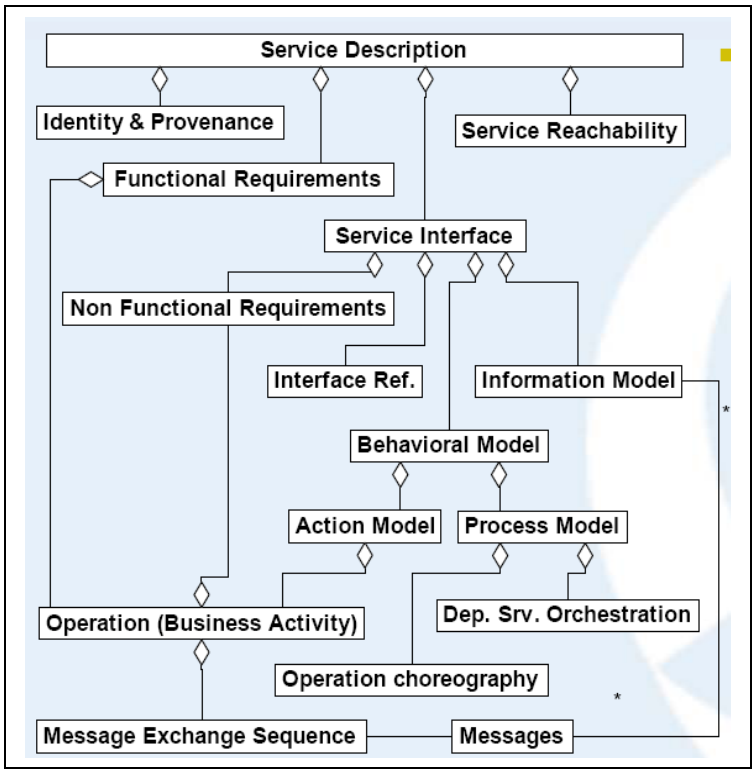


Figure 8: Service description according to mTOP MTOSI

Discussion on SDF

TMF standards - especially eTOM - are widely adopted in the Telecom industry as they provide the conceptual basis for service management activities. They are used in "enterprise architecture" studies (*service urbanism*) aiming to ensure durability of an information system thanks to the elaboration of a target architecture that promotes reuse of services [Simonin10].

The SDF vision shares various ideas with MDA [strassner04], like maintaining clear separation between specification concerns and implementation issues. Now, SDF as defined today remains a *conceptual* framework: it does not propose a specification method or a technical solution for developing services from high level specifications.

2.3.5.2 OMA Next Generation Service Interfaces

Overview of OMA and NGSI

The Open Mobile Alliance is an international organization, developing open, market driven interoperable specifications for global adoption of multimedia and data services. An activity of the OMA is the publication of standards APIs for telecom enablers (such as presence, location, device management, content delivery and so on). The Next Generation Service Interface (NGSI) in OMA can be understood as an evolution of Parlay APIs [parlay] which was designed initially for large scale and major services (like voice,

messages, ring-tones). Actually new revenue potential is in great number of small services demanding specific APIs. For the future, key factors are programmability of next generation services and uniform accepted standards via Open APIs. The NGSI program at the OMA aims at providing the APIs specifications for accessing network capabilities with necessary control for respecting constraints like limited device capabilities, service subscriptions, privacy and user context. NGSI is build upon inheritance of Parlay but has new functional areas to cover like:

- Data Configuration and Management
- Call Control and Configuration
- Multimedia List Handling Extensions
- Context Management
- Identity Control
- Registration and Discovery functions

Discussion on NGSI

Standardization of APIs to access telecom network resources is of major importance for telecom operators since it will facilitate the integration of monetized enablers within the applications developed by third party service developers.
These APIs tend to be specified as REST APIs (and alternatively in SOAP, using WSDL description language). In order to be used within MDA-aware service creation environments (like SPATEL Engine described in Section 3.2.3) it is important to provide a reformulation of these APIs in terms of a modelling formalism like UML. This *refactoring* of the API as a UML interface may also help to eliminate any possible existing implementation dependent elements found in the original specification. This is the kind of work we are doing in TelcoML initiative (see Perspectives in Section 5.2.2).

2.4 Service Development with MDA

Since the core of our study is the use of MDA for the development of telecommunication services, in this section we examine a list of research projects that experiment combination of MDA with SOA-related technology. In line with the topic of our study, our focus will be the development of *composite services* and the development of *interactive voice-based services*.

2.4.1 Selected research projects

2.4.1.1 The COSMO Framework

Authors in [Quartel07] describe a framework for service modelling and refinement. It provides concepts to allow reasoning on services and to assist service designers to perform a variety of tasks such as service composition, discovery and implementation. The

authors define an abstract formalism to describe various aspects of a service. The formalism can be mapped to other specification formalisms (like UML Activity Diagrams, BPML) and can be used to generate implementations (like BPEL, WSDL). An essential characteristic in this work is the focus in the notion of *interaction*. Three abstraction levels are identified: simple interaction (the service modelled as a whole with a requested goal and an offered capability), choreography (multiple related interactions between a user and a provider, modelling the external behaviour of the service) and orchestration (the service modelled from the point of view of a service provider playing a coordinator role). The notation, which is derived from former work named ISDL [Quartel04], is not UML but is looks like a combination of activity and component diagrams, where behaviours boxes contain interactions or actions connected by causality relations annotated with conditions. Information models use ontology based descriptions.

From our point of view, a distinctive value of this work is the simplicity (from a conceptual point of view) of the proposed formalism and its generality that make it usable at different levels of abstraction.

2.4.1.2 UML Sequence Diagrams to WS Choreography

Authors in [Bauer04] propose a translation from UML sequence diagram to BPEL, complemented by - more traditional - class diagram to WSDL transformation. The UML sequence diagrams capture the definition of the sequence of actions between various partners without exposing implementation details like the protocols used for realizing the exchange. Main innovation of this work is the attempt for translating new features of sequence diagrams in UML2 like alternatives and optional fragments, loops and parallel merge.

Lifelines - which represent individual participants in an interaction, are translated by *partner* declarations in BPEL. Receiver and reply clauses are generated for asynchronous messages and for reply messages. Unsurprisingly, alternatives fragments are translated as *switch* BPEL clauses, loops by *while* and parallel fragments by *flow* statements.

While sequence diagrams appear as being very intuitive to provide a general view of the interaction between various participants, it is not adequate to represent stateful behaviour [Micskei10] which may be required when defining the internal service logic of a participant, specially, in our study context, to represent the core logic of a voice based services. In other terms it may be sufficient to represent choreographies but not complex orchestrations.

2.4.1.3 UML activity diagrams for Web Service Composition (UML-S)

[Dumez08] proposes the use of UML diagrams to facilitate the development of composite web services using a specific UML profile named UML-S (UML for Services). UML class diagrams representing elementary web services can be obtained through automatic retro-modeling of existing WSDL files. Then the interface of a composite web-

service is explicitly defined (by means of a UML interface) and the internal behaviour of each operation is modelled by means of a specialisation of UML activity diagram that includes 11 identified flow-control patterns [Aalst03]. At the end a BPEL file is generated to make executable the web-service.

Usage of activity diagrams is very common in projects trying to reuse UML behavioural diagrams to express web service composition (see for instance Section 2.4.1.3). One particularity however is that data transformations between two services invocations are explicitly modelled. Graphically it is rendered as a note listing parameter assignments

2.4.1.4 UML activity diagrams for Semantic Web Service Composition

In [Zhu09], a UML 2.0 Profile is defined to represent OWL-S web services. Class diagrams are used to represent web services where each operation with their declared input and outputs represents atomic web services. Use case diagrams provide information on user interaction attached to one atomic web service and activity diagrams are used to encode web service composition, supporting the 8 basic structures of OWL-S (Sequence, Split, Split-Join, Choice, If-Then-Else, Repeat-Until, Repeat-While, Any-Order). Finally UML constraints are used to represent conditions and tagged values to represent OWL-S categories. An interesting development of this work is the generation of code in a model-checking language called Promela [Holzmann91] to verify the correctness and reliability of the service composition defined in UML.

2.4.1.5 BPMN for semantically annotated Web Services

In [Belouada10], a metamodel representing annotated web services compatible with SA-WSDL descriptions has been defined. In addition a UML Profile is used to allow service designers to specify web services interfaces and semantic annotations graphically. A transformation ensures the generation of SA-WSDL from the UML specification. In the other hand, composite web services are specified in BPMN notation. Then BPMN is translated into BPEL to be executed.

This project adopts a well-known practice of MDA engineering, which is to have a complementary usage of both meta-modeling and UML profiles: the first is for language definition, the second is merely for providing a graphical concrete notation. An important distinguishing characteristic in respect to similar projects (like UML-S) is the choice of BPMN for service composition definition in replacement of UML activity diagrams.

2.4.1.6 Expressing workflow patterns in UML activity diagrams

In [Gronmo04], the authors examine some of the well known workflow patterns - identified originally by [Aalst03] - focusing on those that have a non trivial representation in UML activity diagrams. Then for each of them (Web Service Call, Loop on condition, Data Transformation and Alternative Services) they provide a list of design possibilities and their evaluation. For example, for the Alternative Services pattern (more than one web

service offering the same functionality) the authors provide four design suggestions (like using a fork node with merge or using a unique node with sub-actions) and their preferred solution (the design with sub-actions)

The success criteria provided by authors for selecting the best UML design solution are: readability, completeness for execution and independence of workflow language.

2.4.1.7 UI Modeling and transformation of spoken dialogs

In [Lin09] the authors use a combination of use case, activity diagrams and sequence diagrams to generate an intermediate PSM model containing class diagrams for the three facets of Net-PAC model [Wu02] - presentation, control, abstraction. From the PSM, various code elements can be generated, in particular an executable VoiceXML document to launch the dialog. Automation of the transformation automation is not yet achieved.

From our point of view a noticeable characteristic of this study is the exploitation of PAC model that ensures good separation of concerns. Now the expressivity in the modelling of voice elements is relatively poor (for instance, no sub-dialog definition appears in the methodology). This also comes from the fact that the selected paradigm is not based in state-machines, which is the execution paradigm on which VoiceXML is based.

2.4.2 Model oriented standards for Services

In this section we examine two standards, SoaML 1.0 and BPMN 2.0, recently published by the OMG, that have an important connection with the topic of our work, which is the combination of SOA and MDA. In contrast with standards examined in Section 2.3.2, which are intended to be directly used in an execution environment (like BPEL), the two standard represent are more at the level of service design. The second mentioned standard, BPMN is in fact originally intended for business processes; however, due to the ability to be used for expressing service compositions, we found relevant to examine it.

2.4.2.1 SoaML

Overview of SoaML

The SoaML standard [omg-soaml] attempts to standardize the way UML has to be used to model different aspects of an SOA, like service interfaces, service component implementations, service contracts and policies and so on. SoaML is well-aligned with SOA Reference Model [soa-rm] defined by OASIS. This standard has been officially delivered by the OMG end of 2008, so it is too early to state about its adoption by the industry. Nevertheless it represents an important effort for bridging model-oriented technologies with SOA based technologies, which are generally based on XML. One

important expected benefit will be to bring inter-operability of service definitions at model level.

Since SoaML is very large, we will focus here on selected concepts that concerns the design and implementation phase.

During service specification the service analyser will typically define "service interfaces", which include the offered operations (provided interface) and, when relevant, also the required operations (required interface). To illustrate this, in telecom domain, offering an SMS facility with notification capacity implies for the telco operator publishing not only the interface that clients should use programmatically to send SMS but also specifying the web interface that third parties need to implement in order to receive from the telco operator the notifications of SMS delivering. The ordering constraints for invoking the operations can be specified in more details using other behavioural UML diagrams like sequence charts or activity diagrams.

During realization phase the service designer will typically define the components (participants) that will implement the service interface. This step uses a specialized variant of the UML component notation with service ports referring to the provided interfaces and request ports referring to the required interfaces. The internal behavioural logic can be provided using any behavioural diagram (activity diagrams, state machine) or even using pseudo-code. Figure 9 - taken from the SoaML specification - illustrates the various concepts mentioned above: service interface, participant components with service and request ports:

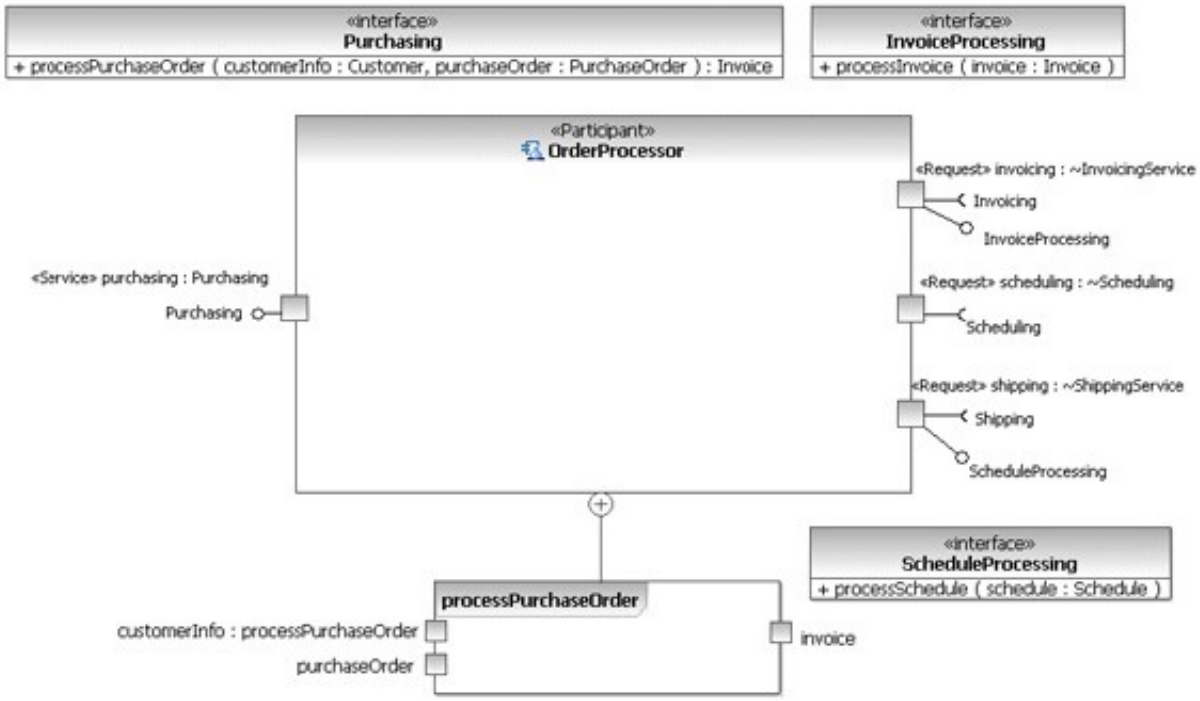


Figure 9: Participant Specification in SoaML

Discussion on SoaML

As pointed by [Kuppuraju09] traditional web service standards (WSDL, SOAP and so on) do not always guaranty inter-operability between solutions provided by different vendors, due to differences in error handling, protocols and versions used. SoaML by raising the level of abstraction, and if appropriately supported by tools, can be a way to make progress on solving these inter-operability issues.

There are however some criticism on SoaML like the way the architecture is modelled (concentrating in relationships between entities instead of modelling the architecture as a whole [Poulin09]).

Being the result of a compromise between various proposals, the SoaML standard attempts to embrace different modelling styles dealing with SOA (capabilities versus interfaces, collaboration diagrams versus component diagrams, and so on). An effective usage of SoaML implies selecting the set of diagrams to be used, possibly taking some decisions - like skipping the modelling of service capabilities to concentrate on service interfaces, or the choice of "document style" for service messages rather than the alternative "RPC style".

Now, from the point of view of service execution, SoaML is not prescriptive at all - different execution paradigms can be used such as Petri-nets (activity diagrams) or state machines, or even other less formal options. This certainly has advantages (let people choose the formalism that best suits to their needs) but may be a problem for people attempting to work with inter-operable executable models - which by the way could be a requirement of agility. This re-enforces the point that to be usable SoaML requires some adaptation.

2.4.2.2 BPMN

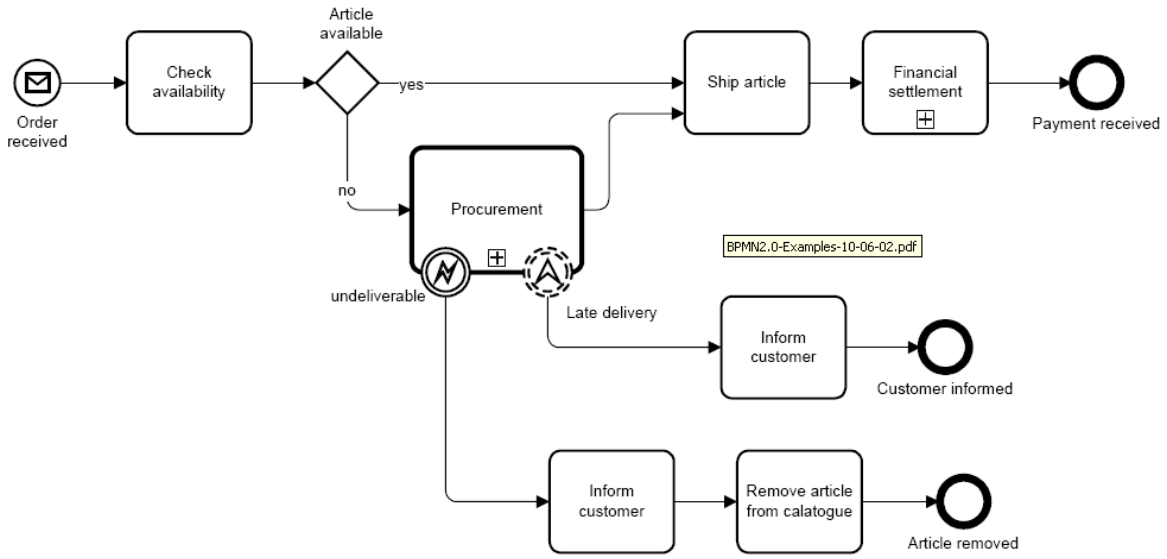
Overview of BPMN

The BPMN (Business Process Modelling Notation) [omg-bpmn] standard defines a graphical notation to model business process that can be understood by business people and IT people. It is primarily a communication tool but, under certain circumstances, could become executable if translated to a formalism such as BPEL. It was originally created by the BPMI consortium and then delegated to the OMG (Object Management Group).

BPMN is similar to UML activity diagrams but has its own notational conventions. There are four categories of modelling elements in BPMN: Flow objects (Events, Activities, Gateways), Connecting Objects (Sequence, Message, Association), Swimlanes for activity partitioning and Artifacts for extensions (like Data, Annotations, Groups). An event denotes something that happens (message arrival, deadline, and so on) whereas an activity denotes something to do. Gateways denotes constructs like conditional decisions, and fork/joins.

BPMN 2.0 has become the new version of the standard since 2010, but is still not widely implemented. It offers some additions like choreography activities and compensation. Most important, BPMN is now defined by a meta-model in addition of being a graphical notation.

For illustration purposes, we provide below an example of BPMN process ("order fulfilment and retirement"). This example is taken from "BPMN by Example" document accompanying the new BPMN 2.0 specification. This example illustrates the case of spontaneous events raised during the execution of an activity - Procument is our case. The *late delivery* does not interrupt the activity whereas *undeliverable* is a fault that stops the activity.



Discussion on BPMN

Heterogeneity in the notations for describing business services causes inter-operability problems. Companies cannot easily switch from one tool to another. In that sense BPML gives the chance to break such dependency. Also, BPML reflects years of experience on business modelling and hence potentially seems well appropriate to help facilitating the communication between analysts and developers, even for complex processes.

Now, since our topic is more service development than process modelling (even if both have some links) we are interested in the capacity of BPMN to represent the logic of composite services, typically orchestrations. The approach that would take BPMN as an orchestration execution language is attractive since this notation is known to be intuitive and easy to understand. However, some extensions to BPMN could be considered in order to be more appropriate for the modelling of service orchestrations, like the ability to

distinguish between service calls and local computations intended for realizing intermediate data conversions.

Indeed BPMN modelling style is not necessarily the best to model complex statefull behaviours - like those we find in interactive voice services.

2.5 State of the Art Conclusions

2.5.1 Summary

The state of the art examines actual practices in model engineering and service engineering and study a list of noticeable research projects that attempts to combine MDA and SOA technologies to develop telecommunication services.

Some of actual research work addresses fundamental issues like recursive modelling of behaviour from one level of abstraction to another (COSMO project) or the analysis of behaviour patterns (in [Gronmo04]). Several other projects put focus on capturing service behaviour specifications that can be automatically translated into SOA aware executable formalisms (WSDL, BPEL). In this topic, multiple approaches are experimented:

In [Bauer04] sequence diagrams are used (among other diagrams) to derive BPEL orchestrations. In [Dumez08] activity diagrams are exploited to reformulate OWL-S composite processes and to generate generates BPEL. In [Zhu09] UML activity diagrams are also used to reformulate OWL-S composite process but the generated target is a verification language to check specification consistency. In [Belouada10] BPMN diagrams represent service logic and in parallel extension to UML class diagrams are proposed to insert semantic annotations.

In the field of voice service modelling, [Lin09] combines GUI and behaviours expressed using activity diagrams to generate applications with VoiceXML based interactivity.

In the SOA standardisation arena, firstly we observe that the standardisation of implementation languages is relatively mature (WSDL, OWL-S, BPEL and so on). In the modelling arena, we see that - thanks to latest SoaML standardization effort - an important progress has been done to propose a uniform formalism for describing static aspects of a service (contracts, interfaces, structure of realization components), while remaining non prescriptive for the behavioural part.

2.5.2 Criteria of research

From our point of view, the following aspects are not well studied in actual research regarding service development driven by MDA:

- Ability to model in an *integrated way* the various aspects of telecom service development: this comprises not only programmatic interfaces and behaviour with or without asynchronous characteristics, but also, semantic definition, non-functional properties and user interaction. In particular the inclusion of voice interaction in the design of "ordinary" services is generally not considered (interactive voice services represent a very specific category of services). Research projects tend to propose solutions that focus on a specific aspect.

- Role of MDA-aware execution frameworks to achieve agility. Most research work dealing with MDA tends to put emphasis in the definition of appropriate formalisms for defining platform independent service specifications. However, solving problems like the substitution of a service component by another equivalent or the construction of context-aware services rely in a large extent in the characteristics of execution frameworks. In other words, the question that arises is: what consequence has the introduction of MDA in the design of modern execution frameworks? Opposed to the traditional view in which MDA simply attempts to map existing middlewares that has no "modelling awareness".

3 Chapter - Contribution

The thesis defended in this report is that an *appropriate* combination of two paradigms, the *Service Oriented Architecture* (SOA) from one hand and the *Model Driven Engineering* (MDE) in the other hand, is the foundation to offer *agility* in the development process of telecommunication services. The areas covered by our study include the case of integrated composite services and the case of interactive voice services. Our contribution include methodology aspects (Section 3.1), architecture and modelling abstractions to support agility for integrated composite services (Section 3.2), architecture and modelling abstractions to support agility for voice based services (Section 3.3) and finally a discussion section, with some recommendations regarding pragmatic appliance of MDA (Section 3.4)

3.1 Approach for achieving agility in development of telecom services

3.1.1 Agility principles for developing telecom services

In the State of the Art part we described the traditional approach taken by telecom operators for developing services (Section 2.1.3.1) as well as the agile manifesto [Fowler01] (Section 2.1.3.2).

We will take the agile manifesto as the starting foundation for developing a methodology for agile development of telecom services. Indeed agile principles are provided in generic way to make it applicable to various areas. Some of them (like P4 and P8 in table of Section 2.1.3.2) are probably too optimistic in respect to the organizational constraints that a large telecom operator needs to manage: for instance a company cannot

afford inviting (and paying) potential users of a new service to talk frequently with developers.

Human motivation and project management aspects as emphasized by the authors of the manifesto are of primary importance for succeeding agility development. We fully agree with this assertion. The focus of our contribution however will be on the *technical means* that we can put in place to help the development process of telecom services to be as *productive* and *adaptive* as possible. Said in other way, our focus remains the tooling - not the team -, understood in its broader sense, which includes not only the concrete software but the abstractions on which it will be based (formalisms, notations, mappings, best practices and so on).

For the specific case of the development of voice based services and composite telecom services, we found relevant the addition of the following three principles to realize agility at tooling level:

X1	Use one or more domain specific languages (DSL) to *specify* relevant aspects of a service in an implementation agnostic manner
X2	Use tools that allow an *immediate* execution of the DSL in a default deployment platform to allow iterative testing and simulation of the service being developed.
X3	Use tools that *automatize* as much as possible the production and the deployment of the service in various execution platforms at terminal and/or server side.

These three principles raise the question of who is in charge for defining the DSLs and for developing the associated tools (adapting pre-existing commercial tools? developing in-house formalisms and the corresponding IDE?). In house development allow companies to get a better control on the evolution of their assets but indeed implies allocating specialized man power to achieve the complex activity of creating an MDA tool chain. In Section 4.2 Evaluation, economic aspects of developing MDA tool chain are examined.

From now, let's comment in more detail our three principles:

Use one or more domain specific languages (DSL) to specify relevant aspects of a service in an implementation agnostic manner: The relevant aspects of a service that would take advantage of a dedicated formalism can be: the service interface (the service operations that client programs will use), the service logic (the expected behaviour, like dialog interaction in a voice-based service or the orchestration sequence in a composite service) and the graphical interface (like the web pages for user interaction or the screens in a native application for smart-phone terminals). Other aspects that would be useful for

service discovery and automatic selection are semantic annotations and non functional features. *Platform agnostic* DSLs are important to let service designers concentrate in their own business rather than being polluted by constraints related to the realization. Ideally such specifications should be understood by business and developers people. Platform agnostic DSLs are also important to allow the automatization (principle X3) of service deployments on different execution platforms.

Use tools that allow an immediate execution of the DSL in a default deployment platform: Immediate execution is essential to eliminate as soon as possible a significant number of errors within a non trivial specification of service logic. In fact very few people is capable of writing large pieces of algorithmic logic (in textual or graphical notation) without introducing errors. Also immediate execution is crucial to let business people validate incrementally a software development, either to figure out weather the requirements are respected, or to help selecting the desired functionalities. Now, by execution we also mean simulation capabilities: depending on the stage of the development some components may not be available when an intermediate version of the service is delivered; in that case the execution tool is expected to offer means to provide a *stub* or simplified execution of the missing component.

Use tools that automatize as much as possible the production and the deployment of the service: the effort provided for producing precise executable specifications should be rewarded by some significant productivity gain. In the case of services that requires a deployment in multiple devices (PC, various kinds of smart-phones, TVs, and so on), the potential productivity gain obtained thanks to automatic generation can be very high. Now, this implies having an appropriate strategy to deal with extra platform-dependent information that is generally required to generate code for specific platforms (annotating the DSL model? Creating an implementation model?). Also this implies managing the level of variance and flexibility that someone may want to introduce to the generated production (like, what is I want to select a different presentation style for the generated interface?).

To conclude, in attempting to make agile the development of telecom services, we see that generative techniques will have a major role. In fact, if we refer to one of the agile value statements "focus on software rather than in documentation", our generative approach is seeking for reconciling both by making "the specification play the role of software".

We should point out that the approach we are presenting here share some of the fundamental principles of the *host-target development and testing strategy* used traditionally for the development of embedded systems [Ipl96]. In this strategy the software is developed in a different environment than the environment in which it will eventually be used, the development environment is the *host* and the final environment for execution is the *target*. A popular illustration of this paradigm is the GCC cross compiler producing binaries for different operating systems [gcc]. In our case, however, generally the *host* also represents a *target*: hence it is more than a testing environment; it is an effective *native target* to run service logic. Differences also concern the context of usage of these general host-target principles. Firstly we are explicitly promoting usage of object oriented *models*

(OMG meaning) - rather than using general purpose languages with portable libraries, and secondly, we are going to apply it to a domain, telecommunication service development in our case, that will have various specificities depending on nature of services to be developed - like voice-based services or composite integrated services. This means that an important effort will need to be put on identifying specific methodological steps (like in Section 3.1.3), finding the appropriate constructs at the language level (like in Section 3.2.2 and 3.3.1) and appropriate features at execution framework level (like in Section 3.2.3 and 3.3.2).

3.1.2 Realizing agility with model-driven technology

3.1.2.1 Rationale

In the previous section we have presented three principles for agile development of services that emphasizes the role of automatic code generation. To realize this vision our first choice will be the usage of *model-driven* techniques: the service specification (the DSL) will be a *model* and we will apply to it *model to model transformations* and *model to code* generations.

However it is important to point out that a generative process may be realized using other technical means [Emmen02]. For instance to define DSLs, we could:

- Create XML documents that follow a manually-defined structure, even without formal XML schemes.
- Use a general purpose language to directly declare metadata (for instance by means of associative arrays in configuration files). The *Django* framework [django] for web development [Django] has an interesting approach: models are explicitly defined by Python [python] classes and are exploited to generate important parts of the application.

There can be also some practical considerations that can make people not use model-driven technologies in their projects (see discussion in section 3.2). However, from a conceptual point of view model-driven formalisms contain all necessary ingredients to realize the vision in a *clean* way [omg-mdag]. It is worth to mention that a model is not only an artefact that can be manipulated by a machine to realize automatic tasks like conformance checking and code generation, but also represents an abstraction that helps people to clarify ideas [Guerbi09].

3.1.2.2 Exploiting model-driven formalisms for service development

In reference to the three agility principles, when model-driven technologies are used:

- DSLs are defined conceptually by MOF meta-models. A metamodel has a machine-readable format in XMI and can be visualized using human-readable UML class-diagrams.

- Service specifications are represented by models conformant with the DSL metamodels. These models can be serialized using XMI, but can also be provided using one or more concrete syntaxes, such as a graphical one based on UML diagrams or another textual notation. In some cases, the mapping between concrete and abstract syntax can be defined formally, although, this is far from being an easy task [Milanovic09].

- Transformations (model-to-model and/or model-to-text) can be defined using directly the concepts of the DSL. These transformations can be written in a variety of ways (see Section 2.2.2.3), for instance using QVT [omg-qvt]. Whatever is the technique used, working directly with DSL concepts, avoids syntax pollution and low-level manipulations, like when dealing with XSLT transformations [w3c-xslt] [Duddy03].

3.1.2.3 Scope of automatic code generation for behavioural specifications

The main promise of model-driven engineering on helping to achieve agility is that it minimizes the gap between the specification of the service and its implementation [omg-mdag]. The "easy" part of code generation is to translate structural definitions, like converting a service interface definition into a couple consisting of a complete Java interface and a skeleton of a Java class. The more difficult part indeed is to deal with the translation of the specification of the behaviour (the service logic) [McNeile03].

Regarding behaviour specifications, during our experiments we observed three typical situations:

- Either the specifiers only want to provide an "idea" of the sequence of actions to be performed. In that case it is purely documentation stuff, and it cannot be exploited by code generation.

- The specifiers really want to provide the details of one part of the logic, while possibly skipping details of other parts. In that case code generation can be applied but some conventions need to be taken to deal with the undefined parts. In addition strategies need to be defined to handle the combination between generated code and manually-written code.

- The specifiers provide all the details of the logic. In that case code generation can be fully applied without the need for managing manually written code.

In the case of *explicit* service compositions (expressed for instance in BPEL, or in UML activity diagrams or any ad-hoc *mashup* formalism), we are more in the second case: the logic is intentionally specified and represents an orchestration of services. However some internal calculations - like data conversions - may remain opaque and then be implemented in the target general purpose language. For this category of services we

developed the SPATEL DSL and corresponding SPATEL Engine framework (see Section 3.3).

In the case of voice applications, in most cases, we are in the last situation: the focus is the detailed specification of the dialog interaction between the user and the voice machine. State machines can be used to fully capture this interaction logic, hence all the part dealing with this interaction can be generated. The Voice DSL and the associated Voicebench framework represent our proposed solution for this kind of services (see Section 3.4).

In between, we have many other categories of services. Some of them, called *service enablers*, play a central role in telecom because they expose strategic functionality of the operator to third party developers, like address book, identity and messaging. The service enablers are generally described as basic services (non composite services). Other services are not originally described as an orchestration of other services; however, after examining them we may found that they could be reformulated in such a way, and hence take advantage of the tooling developed for composite services. We call these services *implicit* service compositions. For sure, conceptually almost all services could be realized as a composition of other services. The question is more to know weather such description really helps when implementing the service. This is typically a design decision. In any case, our contribution regarding the exploitation of behaviour specifications focuses in voice services and in services that can be reasonably formulated as composite services.

3.1.3 From the idea of a service to its realization

In this section we describe our practical vision of an *agile process* for developing integrated composite services (mixing telecom and IT facilities). Our starting hypothesis will be that the service in question is a service that can be understood as a composition of other services. Our goal is not to invent yet another agile method in its wide sense: firstly, we intentionally skip resource management aspects (team motivation, organization, and so on) to concentrate on tooling aspects, secondly, it is dedicated to a specific domain, thirdly, we believe it can be incorporated in most of the existing methods. In our case, taking into account actual practices in France Telecom, we will describe it as a specialization of the RUP software development process [Kruchten99].

3.1.3.1 Life cycle phases for service development

In order to develop a service, there is an initial phase in which some people try to make explicit what they have in mind (functionalities) and what are the constraints to be taken into account (non-functional features). This phase may or may not be strongly formalized depending on the organization and context (research initiative? service request from an operational business unit?). If we refer to the RUP software development process, this phase corresponds to the *inception* life-cycle phase.

After that, people need to think how to organize the work, which means identifying modules that need to be developed from modules that exist and can be reused, as well as the more appropriate sequencing of tasks. This phase is something that is typically performed by actors playing the role of "software architects". Indeed software designers or even software developers can play this role in the case of teams with a small internal organization - as recommended in some agile methods. In RUP terminology this corresponds to the *elaboration* life-cycle phase.

The next phase is the *construction* phase which in our case includes the detailed specification of the service, transformation appliance and multi-target implementation. Various *design* decisions will be reflected in the transformations, like the structure of the generated code. We do not dissociate specification, design and implementation because agility demands having close relationships between them, especially due to the need of incremental and iterative work and the exploitation of code generation techniques. This phase comprises early executions and simulation of partially implemented service operations. An *immediate* execution of the service after each iteration can be made possible thanks to the availability of a *native framework* implementing the DSLs (see SPATEL Engine in Section 3.3 and VoiceBench in Section 3.4).

The last phase, following RUP, is the *transition* phase, which is in charge of putting the developed service in production. This involves indeed more formalized testing than what was done in the previous task. Automatic generation of test procedures may provide a significant productivity gain (see an example in section 3.3.2). It implies also re-applying deployment procedures already done in the previous phase, but targeting production environments rather than development environments.

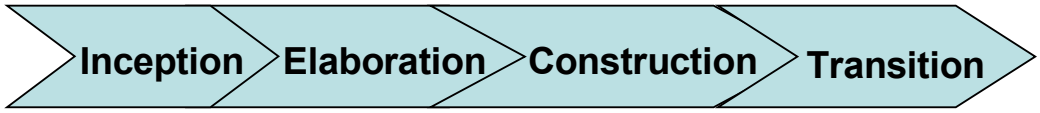


Figure 10 Typical Life-cycle phases

Below we describe in more detail the *elaboration* and *construction* phase.

3.1.3.2 Elaboration phase

Following the well-known paradigm *divide to conquer*, the main idea is to try to describe the intended service as a *composite component* with possibly more than one level of depth (where a part may recursively be a composite). Another important consideration is that we need to distinguish between three things:

(i) the core logic of the service containing *internal* components,

(ii) external *friend* components invoked by internal components in the core,

(iii) interaction interfaces (GUIs, received asynchronous events) used to activate and access the service.

In the most common case the core logic runs in an application server, whereas the interaction interfaces run on terminals as web applications or possibly native applications. *Friend components* will either run at the server side (for instance an SMS messaging feature) or at the terminal side (for instance the GPS geo-localization component).

Based on these concepts, the process for defining the architecture of a service will be:

- A1: Decompose the service as a tree of components, where some are considered internal (to be developed) and others are considered friends (possibly pre-existing services, or standalone *basic* services to be developed). A basic component is a component that is seen as a black-box component, hence has no explicit decomposition.

- A2: Define informally some relevant interactions between the components: order of messages, actors, and so on.

- A3: Define the iterations to be executed: Not all service logic implementations will be achieved at the beginning. In an early iteration an internal composite service may for instance be seen temporarily as a basic service (with no exposed decomposition).

For the elaboration phase, different kinds of diagrams can be used: For task A1, for instance, a simple informal tree representation with nodes and leaves to represent internal and friend components, or a richer UML component diagram. For task A2 a list of UML sequence diagrams. In fact we consider the elaboration phase as a period that is useful to clarify ideas and establish mutual comprehension. This phase may be heavy or really light depending on organisational constraints.

3.1.3.3 Construction phase

In essence this phase will be concerned with creating the detailed service specification, apply code generation and achieve the complete implementation. The phase is heavily dependent on the availability of productive tools. The service will be formally specified using a DSL. The hypothesis we made is that a service is roughly defined by a service interface containing service operations, where service operations may have or may not have an explicitly defined logic.

The following typical tasks are to be performed, taking as input the decomposition produced by the elaboration phase:

- I1: *Specification task*: Define the service interface of each component, needed by the iteration being executed. At this stage, the interface of a friend (external) component may be adapted to the needs of the new service. In that case it becomes a *mediation* service. One motivation for adapting an existing interface to make it as *neutral* as possible is to facilitate service substitution in case of changes in the environment (see Section 3.4.2.1 *Enabling vertical and Horizontal variability*).

- I2: *Initial Implementation*: Implement each of the components for the "default" platform. In practice this includes the behaviour logic of service operations, possibly complemented with GUI aspects. In some cases the implementation can be a *stub* implementation, which is an implementation that exhibits a simplified behaviour - may be nothing at all - in respect to its expected final behaviour. An implementation can be derived automatically in the case the behaviour of the service operation is explicitly modelled (like for explicit service orchestrations). Otherwise a skeleton of code can be generated to speed up a manual implementation.

- I3: *Simulation*: Immediate execution of the service each time a component implementation reaches a stable situation. This is permitted by the native framework supporting the DSL in a default execution platform.

- I4: *Multi-target Implementation*: Firstly, complement service interfaces and service logic with information useful for generating alternative implementations of the service or alternative access interfaces (such as a mini-application for each of the most popular smartphones platforms: the IPhone, Android and Nokia S60). Secondly, perform automatic code generation and any required manual completion to realize the alternative implementations.

- I5: *Publication*: Some components implemented in I2, will be promoted for reuse. Typically they will be published as standalone web services.

The construction phase we described above is, as the whole global process, iterative and incremental. In particular, errors found when applying specific code generators (task I4) may question the complementary information added in task I3 or even the initial service interfaces in task I1.

Section 3.3.3.4, describes the artefacts that are involved in the construction phase when developing services with the SPATEL Engine.

3.2 Composite Services: SPATEL and SPATEL Engine

3.2.1 Introduction

Service composition has become a hot topic for all telecommunication players. The ability for professionals and, even more for end users, to compose efficiently running

telecom components, depends a lot on the availability of tools capable of hiding the complexity to access the telecommunication network resources. Many initiatives are currently launched in the telecom arena to try to solve the complexity of distribution and heterogeneity, especially now that the operators tend to open their access to their network resources, like the *Orange Partner* program for 3rd party developers [orangepartner].

In this section we present our proposed solution, which consists essentially of two elements: the SPATEL domain specific language and the environment build on top of it, including mainly the orchestration engine called SPATEL Engine. We emphasise here the importance of having both artefacts in place (the DSL and the framework that exploits it) in order to realize the vision of agility in service creation presented in Section 3.1. A specific sub-section treats methodology aspects when dealing with services developed with SPATEL and SPATEL Engine, in connection with the agile process presented in Section 3.1.4.

Notice that detailed uses cases using the SPATEL formalisms are described in the Validation chapter (see 4.1.2 and 4.1.3).

3.2.2 The SPATEL language

The SPATEL language allows the specification of various aspects of a service such as the service interface, service operation logic (like the orchestration logic of a composite service), voice interaction dialogs, GUI features, semantic annotations (inputs, outputs, pre-conditions, effects and goals) and non-functional properties of services.

SPATEL allows service compositions to be represented using state machines, hence enabling the formulation of complex interactions in the orchestration, possibly involving asynchronous communication and long-running behaviour. Once SPATEL specifications of available services are published on a registry, these services can be discovered, selected and used in service compositions. Using SPATEL and the related *service creation environment* (SCE), service developers can compose a new service made up of an orchestration of different services, typically running in different service providers' domains.

The SPATEL language is platform independent in the sense that it allows defining service interfaces and service logic in a technology-agnostic way: no assumption is done concerning the used execution engine and the communication protocol to actually deploy and run the described services.

One of the objectives of the SCE to be build on top of SPATEL was the idea of supporting the developer to discover services matching a particular goal and being able to suggest compositions of existing services to realize the goal. In order to achieve some degree of automation, semantic annotations must be added to service descriptions. As dynamic composition of services is based on service semantics, it is essential to provide mechanisms to semantically annotate new and already existing services.

The SPATEL language is technically defined by means of a meta-model [omg-mof] from which a programmatic API and XML machine-readable serialization are derived. Associated to this metamodel there are two concrete notations for the users of the language: a pure textual notation and a graphical notation based on a UML profile [omg-uml]. Depending on the kind of users of the language, one of the two offered notations can be preferred: the graphical one is particularly suited to collaborative work between service designers, but it is often less scalable than the textual notation when formalizing a complex service logic.

A noticeable particularity of SPATEL is support of a complete expression language thanks to the inclusion of Essential OCL [omg-ocl] as expression language. This is an interesting feature to ease integration with modelling environments that already have OCL support incorporated. This also has the advantage that OCL iteration operators (like *collect* and *forAll*) can be used to make behavioural specifications with complex condition expressions more concise.

The complete metamodel and grammar definitions are provided in annex B (Sections 9.1 and Section 9.2).

In the following sections we focus on essential concepts of SPATEL.

3.2.2.1 External view of a service

A service in SPATEL is primarily described as a black-box interface which provides the information service clients typically require to operate with it. This service interface declares a list of operations, a list of input and output events, multimedia streams and relevant side-effects. The constraints on the usage of a service interface such as the ordering of operation invocations can be precisely defined through a contract specified by means of a UML sequence diagram.

Figure 11 below shows an excerpt of the SPATEL metamodel depicting main external view concepts. The central concept is *Service Interface* that consists of provided *Service Operations* and exposed *Service Attribute* for configuration, as well as declared service contracts. Other specificities of the external view is the ability to declare, attached to service operations, emitted or received asynchronous events, streams and even side effects.

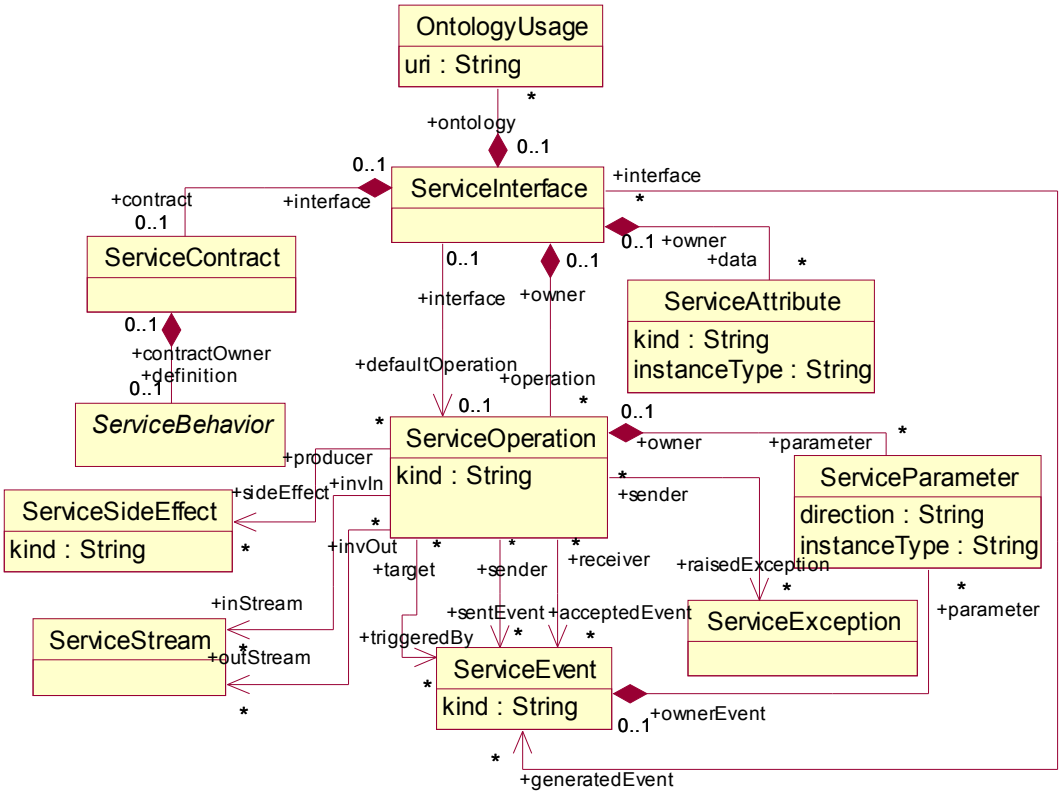


Figure 11: Excerpt of SPATEL metamodel

The example below shows a service interface definition for a simple multi-protocol messaging service (involving email, instant messaging or SMS). There is one operation for each mode of communication and a generic and context-aware 'send' operation (the selection of the messaging done is done at run-time on basis of detected user presence).

```
service Messaging {
  sendEmail(receiver:String,subject:String,
            msg:String): status:String;
  sendSMS(receiver:String,subject:String,
            msg:String): status:String;
  sendIM(receiver:String,subject:String,
            msg:String): status:String;
  send(receiver:String,subject:String,
            msg:String): status:String;
}
```

The graphical view of this interface is depicted by Figure 12 below:

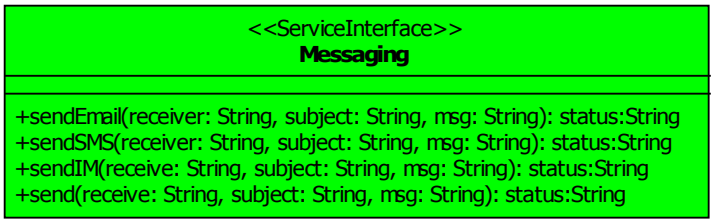


Figure 12: Service Interface for a multi-protocol Messaging service

The operations and parameters of a service interface can be annotated semantically in order to allow scenarios dealing with service discovery and service composition. A detailed explanation concerning this aspect is provided in Section 3.2.2.5.

3.2.2.2 Internal view of a service

In addition to the external view described above, the SPATEL language also allows describing the service as a white-box, that is exposing a partial or complete specification of its internal behaviour. More precisely, the logic of a service operation in the interface can be defined as an orchestration - a centralized composition - of other services. In contrast with more "traditional" request/response services found on the WEB, a service operation in our telecommunication context may be long-running and have its execution being stopped waiting for the arrival of asynchronous event notifications. The paradigm used in SPATEL to support this kind of behaviour is state-machine based. State machines are particularly useful to represent complex interactions typically used in voice dialogs or in multi-modal services executing in mobile phones. Figure 13 shows an excerpt of the SPATEL meta-model dealing with service logic (this part of the SPATEL metamodel is essentially a simplification of the UML state machine metamodel representation, except for the detailed kinds of actions and events which are specific to SPATEL).

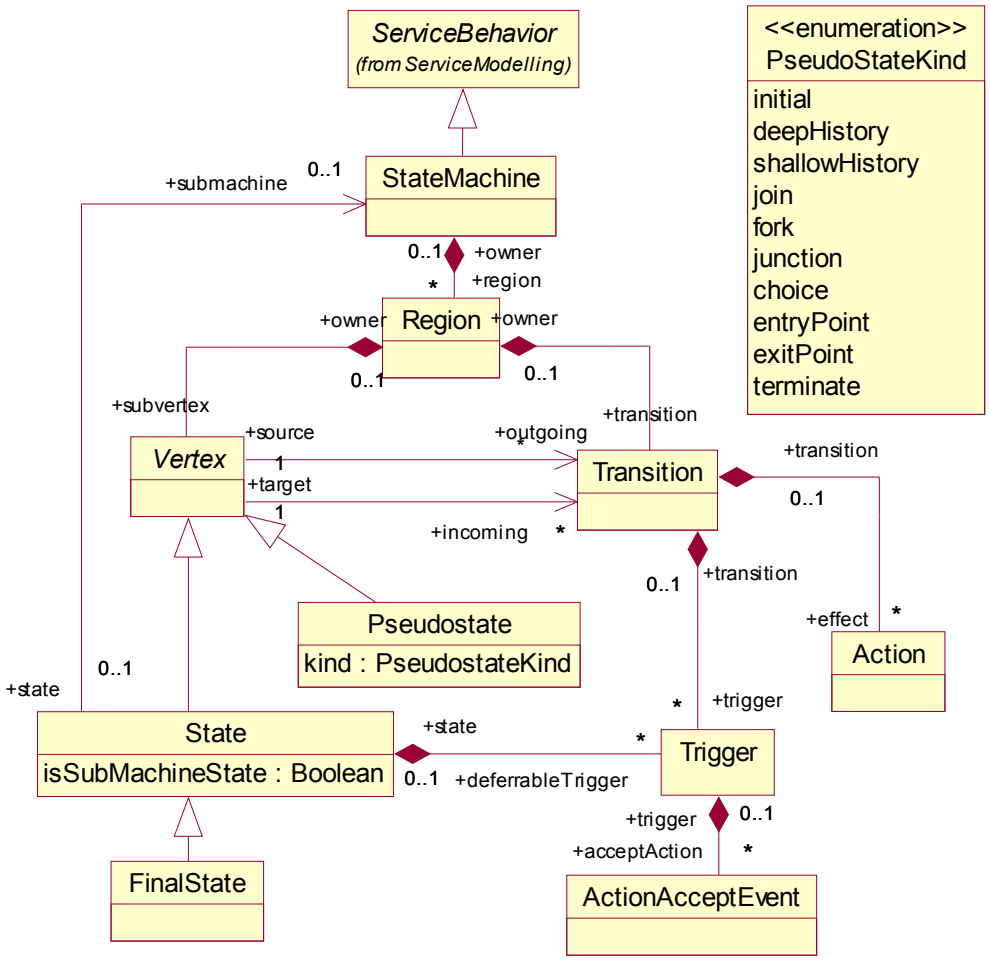


Figure 13: Excerpt of State Machine abstract representation

Figure 14 shows the category of actions that can be attached to transitions. We can note the presence of an *uninterpreted* action, which may be useful for simulation of incomplete behaviour specifications. Also various voice-interaction specific concepts are defined, like the Play action to vocalise a message (text to speech).

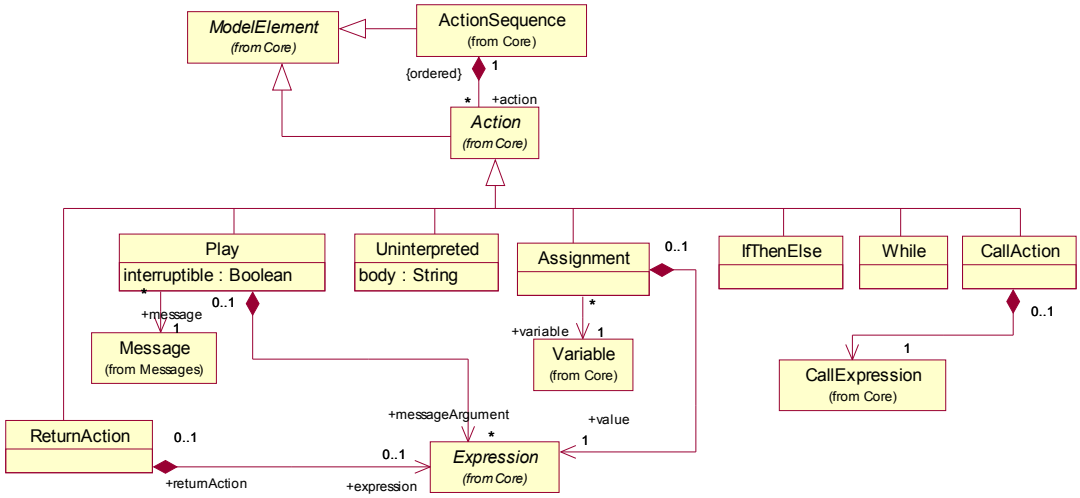


Figure 14: Kind of actions

In the example below, we provide a specification of the logic of the generic 'send' operation of the Messaging service interface depicted in Figure 12. Basically it inspects the user profile and checks its *presence* status to see whether an instant message or an SMS has to be sent. In the end, an email report is sent. The login identification step is required depending on user preferences. We provide here the version in textual notation.

```
behavior Messaging:: send (receiver:String,
   subject:String,msg:String) : status : String {
using PROF:UserProfile, ENV: EnvironManager,
      PRESENCE: PresenceManager;
state Start:
  var login := PROF.getLogin();
  if (login.isEmpty()) {
    transition -> PwdRequired;
  } else { transition -> PresenceTesting; }
waitstate PwdRequired:
  accept {
    on (LoginEvent(login) ) {
       if (PROF.checkLogin(login)) {
          transition -> PresenceTesting;}
       else { raise ErrorLoginFailed(login); }
    }
    on (Reject) {
       raise ErrorLoginRequired(login);
    }
} }
state PresenceTesting:
  var available:= PRESENCE.checkAvailability(login);
  if (available) {
    this.sendIM(receiver,subject,topic);
  } else {this.sendSMS(receiver,subject,msg);}
```

```
state Reporting:
  this.sendEmail(receiver,"Notify report",
       ENV.getDate().asString()+subject+"\n"+msg);
}
```

In the example the logic traverses one waiting state (PwdRequired) and three transient states (Start, PresenceTesting and Reporting). Three *friends* components are composed - they are introduced by means of the **using** keyword. The instance PROF of type *ProfileManager* is used to access user profile, the instance PRESENCE of type *PresenceManager* is used to check availability of the destination user and the ENV instance to retrieve the date of the day.

3.2.2.3 Voice dialog modelling in SPATEL

In the service behaviour example presented in previous section, the provision of the login information (within the PwdRequired state) is represented by the reception of the *LoginEvent* signal. An important point is that nothing in this code reveals how this event is actually generated. Here is where some light-weight form of multi-modal service design can be introduced. In fact, it could be generated by different means such as through a GUI or through a voice interface.

A typical code generator applying on the behaviour specification of the 'send' operation, will detect that user interaction is required when reaching the PwdRequired state and will produce consequently automatically a graphical interface that allows an end-user to enter the login information. In parallel to this, the service designer may want to add support for retrieval of login information using voice, as an alternative mean of authentifying the end-user. This second way of interaction would be useful in mobility situations where hands cannot be used. Upon successful check of the spelling of the response, the system retrieves the stored login information - and hence generates the *LoginEvent* signal that is expected in the service logic.

The specification of the voice interaction to retrieve the login is specified through a *dialog specification* that complements the previous behaviour specification. The formalisms for dialog modelling in SPATEL were taken from the Voice DSL presented in Section 3.4 - which is a more specialized language for defining voice-interactive services. An excerpt of the definition in textual format is the following:

```
  dialog GetLogin(receiver) generates LoginEvent {
    play NameOfYourMotherMessage();
    accept
    on NameOfYourMother() {
      var st:= PROF.checkQuestion("MotherName");
      var login :=  PROF.getLoginInfo(receiver);
      if (st) send LoginEvent(login);
```

```
    }
  on Inactivity() { …}
  on Reject() { … }
   …
}
```

The example presented here shows one of the possible patterns of inter-leaving between the use of voice and GUIs together. More sophisticated and complex synchronization may be needed. In general the level of granularity where these two modes of interaction can coexist within a service logic defined in SPATEL is the state. A state can be either augmented or overridden to add or to hide behaviour, respectively.

3.2.2.4 GUI support in SPATEL

Another important characteristic of the SPATEL formalism is the ability to specify some details of a GUI, typically needed at terminal side to provide the required inputs expected during the execution of the service logic running at "server side". A GUI definition in SPATEL can be seen as an assistant to code generation: the generated GUI will be influenced by a hierarchical description of the GUI resources. Such GUI definition is optional. If not present a code generator applies default settings to generate appropriate buttons for starting or sending intermediate asynchronous events to the service.

GUI support in SPATEL is not intended for generating complex graphical interfaces, since the scope remains service development and not application development. Various research work concern the generation of complete web applications using models [Moreno07]. In our case, we only needed to exploit structural GUI information as well as a connection between GUI events and service events.

The metamodel part dealing with the GUI adds four concepts: UiContainer, UiElement, UiProperty, UiEvent and UiTrigger. We provide below its formal definition (in QVT/EMOF textual notation). The complete SPATEL metamodel is provided in Annex B.

```
class UiElement extends Variable {
  kind : String;
  composes attribute : OrderedSet(UiProperty); // [*],[1]
  composes ownedEvent : OrderedSet(UiEvent); // [*],[1]
  composes trigger : UiTrigger; // [0..1]
}
class UiContainer extends UiElement {
  composes element : OrderedSet(UiElement); // [*],[1]
}
class UiProperty extends Variable {
  value : String;
  linkValue : UiElement; // [0..1]
}
class UiEvent extends ServiceEvent {
  bindExp : String;
  bindTo : ServiceEvent; // [0..1]
```

```
  }
  class ServicePackage extends ServiceLibrary {
  }
  class ServiceClient extends ServiceElement,Package {
    composes ui : UiContainer; // [0..1]
  }
  class UiTrigger extends Trigger {
    composes effect : ActionSequence; // [0..1],[0..1]
  }
```

To deal with heterogeneity of widget systems existing in mobile environments, the approach taken in SPATEL is to have a "generic" coding schema that avoids inventing a new model or favoring a specific one. In the SPATEL metamodel, a GUI Container contains recursively GUI Elements which in turn define GUI properties – which are name/value pairs. Moreover, GUI events can be connected to service events used within the logic of the service. An example of a configuration for a GUI is depicted below. Notice that the concepts in use, such as Label, TextField, Button, PhotoAlbum are explicitly exported from a library named "simplewidgets". Other widgets libraries could be used.

```
userinterface FlickrTag::main uses "simplewidgets" {
   Ui _ui {
     attribute flex = "1", title = "FlickrTag";
     Group _group {
       attribute kind = "hbox", align = "center";
       Label _label { attribute text = "Tags :"; }
       Textfield tag_field {attribute flex = "1";}
       Button push { attribute text = "Go"; }
     }
     Photoalbum album { attribute flex = "1"; }
  }
}
```

Supporting a GUI framework - like the one provided in Symbian S60 environment [symbian-s60] - means two things: (i) having the corresponding library of widgets components instantiated in the SPATEL design tool, and (ii) having the corresponding code generator targeting the specific GUI framework.

3.2.2.5 Semantics and non functional annotations

In order to support reasoning on service features, the SPATEL language provides a generic mechanism for adding semantic annotations and non functional features to a service specification. Their main purpose is to help the discovery of services (at design time or at runtime) and to enable scenarios where dynamic composition is needed. In Section 3.2.2.5.3, we provide details how the annotation mechanism is formalized in the SPATEL metamodel.

SPATEL service descriptions contain semantic annotations in the form of references to concepts of a given ontology, defined in RDF [w3c-rdf] or OWL [w3c-owl]. Referenced ontologies define a taxonomy of concepts enriched by semantic relations between nodes; each concept is defined as a subset of its parent(s) and conditions can be specified as formal restrictions over its parent following a Description Logic (DL) formalism.

A specific telecom-oriented ontology named the Mobile Ontology [villalonga] was defined and used in SPATEL descriptions to annotate services for experimenting automatic discovery and automatic composition facilities. The Mobile Ontology is structured in sub-ontologies to cover different domains: NF-Props, IOTypes, Goals, Service Context, Profile, Presence, Context, Distributed Communication Sphere (DCS), Content, and Privacy.

If we take the Messaging service example, presented in Section 3.2.2.1, the following declaration (in textual notation) will complement the definition with semantic information:

```
service Messaging
   using ontology MobileOntology
         ("http://www.spice-ist.org/MobileOntology");
semantic Messaging::sendSMS {
   GOAL -> MobileOntology::SMS;
   receiver -> MobileOntology::PhoneAddress,
   subject -> MobileOntology::MessageSubject;
   msg -> MobileOntology::MessageContent;
}
```

The semantics block contains a list of semantic relationships: the 'receiver' parameter is for instance connected to the PhoneAddress semantic concept: this means that receiver parameter (represented by a generic String) is actually a more specific data type (the PhoneAddress) whose semantic meaning is formally defined within an external ontology XML document.

Figure 15 gives a richer example of an annotated service (a flight booking service) in SPATEL graphical notation. The service contains three operations (SearchForCheapestFlight, BookFlight, CancelFlightBooking). Different kinds of annotations are used in this example (goals, effects, pre-conditins, QoS, and so on). See Section 3.2.3.2 *Annotation Types* for their meaning.

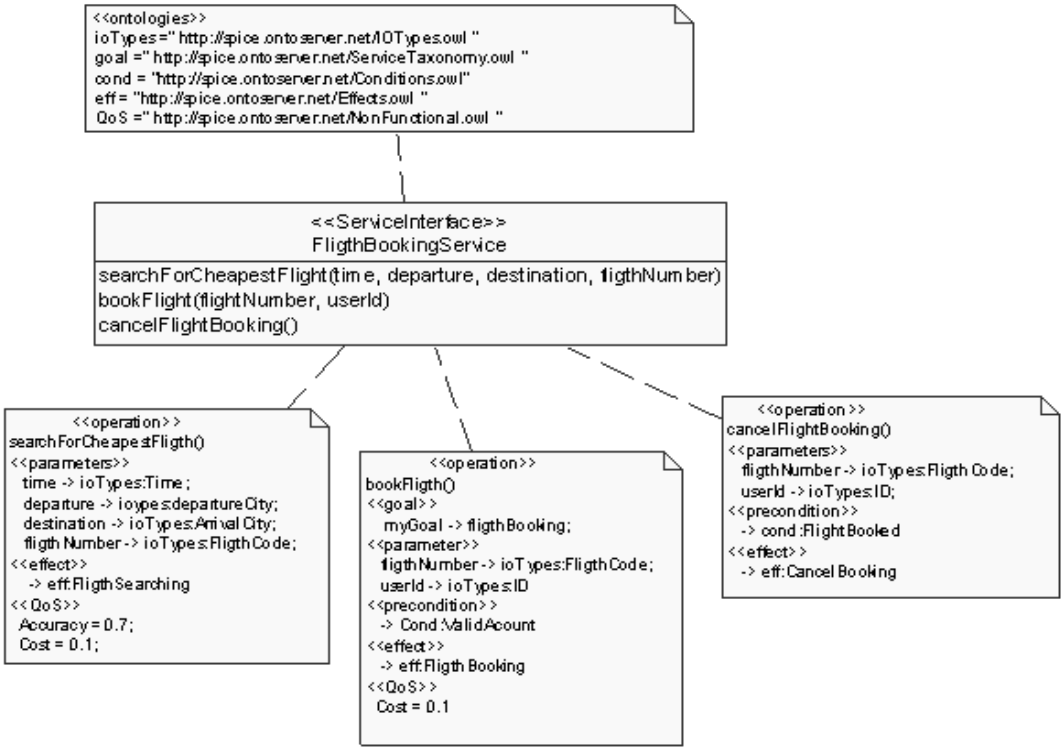


Figure 15: Flight Booking Service Example

3.2.2.5.1 Patterns for semantic and non functional annotations

Different strategies can be used to semantically annotate services in order to enable dynamic discovery and composition. Annotations may depend on the kind of services (stateless or stateful), their intent, and the technology used to invoke them. In SPATEL, two patterns are explicitly defined for semantic annotations: GQIO and GQIOPE.

The GQIO (Goal/QoS/Input/Output) pattern focuses on the core semantics of a service by specifying its goal, the semantic type of its parameters, but it does not require non-functional properties. It is particularly suitable for stateless services with simple operations. Annotations on the goals could reside both at the service and at the operation level while the others reside only at operation level.

The GQIOPE (Goal/QoS/Input/Output/ Precondition/Effect) pattern adds annotations on preconditions and effects of the service operations. It is particularly suitable for complex services (possibly stateful) since these additional annotations allow one to formally specify the functional dependency between operations.

3.2.2.5.2 Annotation Types

Different kinds of semantic annotations are present in SPATEL, namely:

- Annotations on *input/output* parameters refer to a given parameter and describe its semantic type (i.e. arrival time or number of tickets). They allow to build chains of service components by comparing service operations and matching the semantics of their input and output parameters in order to finally assemble them. Annotations on parameters are necessary in order for a auto composition tool to be able to match services and adapt inputs.

- *Goal* annotations describe the overall objective of a service (i.e. goal:FlightBooking) and/or the specific objective of an operation (i.e. goal:CancelBooking); they enable semantic service discovery. Annotations on goals could reside both at the service level and at the operation level. An operation without a goal annotation implicitly assumed that its goal coincides with the service one (that in this case must exist). Whenever a service is composed by several operations and each operation has its own sub-goal, the overall service goal (if exists) constitutes the functional context in which the different operations should be interpreted.

- Annotations on the *effects* of a given operation describe the outcomes of its execution in terms of state achieved by the service or action performed; therefore their scope is bound to a single operation.

- Annotations on the preconditions of a given operation describe the conditions that have to be satisfied in order to allow its execution; therefore their scope is bound to a single operation. Common preconditions could relate to the user profile (i.e. credit account) or the context of use (i.e. terminal used). It is possible to have multiple pre-conditions for a single operation, and they are interpreted as a conjunction (i.e. logical AND) of many conditions.

- Annotations on non-functional properties describe aspects related to the quality of service, charging or resource usage. Such semantic annotations allow filtering and selecting services on the basis of their performances and QoS. The scope of such annotations is related to an operation and their use is optional.

3.2.2.5.3 Annotation mechanism in the SPATEL metamodel

Figure 16 shows the part of the SPATEL metamodel that defines how semantic and non functional properties are encoded.

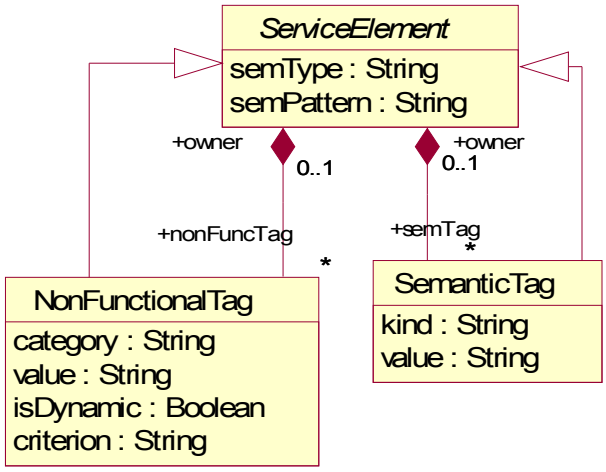


Figure 16: Annotation mechanism in SPATEL

Excerpt of SPATEL metamodel: Metadata for semantic annotations.

A *ServiceElement* is a generic concept representing all service model elements. This means that all service model elements own potentially the properties defined for ServiceElement. We describe below the usage of each property.

The *semPattern* is used on *ServiceInterface* instances to declare the semantic pattern being used; the semPattern is a generic property of ServiceElement to allow the possibility to override the semantic pattern in a sub-element of a service interface. The two pre-defined values are "GQIOPE" and "GQIO". Other patterns values could be defined by service designers to take into account other needs.

The *semType* property is used to reference a node in an ontology. It is typically used in a service data type to refer to the semantic type definition or in a service parameter to indicate the semantic type. It is also used in a goal annotation to refer to the ontology node representing the goal.

For a semantic tag, the kind field represents the tag type. Examples are "goal", "effect", and "precondition". The value property is used whenever the referred node needs to be characterized with a value.

For a non functional tag, the category field represents the general set of a non functional property. For instance "QoS" or "Charging", while the criterion defines the actual metric to represent a property, and the value field gives its expected value, for example:

category="Charging", criterion="cost", value="0.1"

The *isDynamic* slot set at *true* informs on the possibility that the value is actually computed at run-time (thus its value dynamically changes), while a *false* value means that the value is set at service design time.

3.2.2.6 Summary on the SPATEL language

Different opportunistic approaches are taken to specify the different aspects of a service: the voice part uses a dedicated sub-language, whereas the GUI part uses a generic representation approach (based on the availability of widgets libraries, which can be added on-demand to the design tool).

To conclude with the description of SPATEL, we can say that the SPATEL formalism basically aggregates well-know constructs coming from different sources (VoiceXML [w3c-vxml], ITU-SDL [itu-sdl], SA-WSDL [w3c-wsdl] and UML [omg-uml]) in order to provide features needed for a high-level and executable formalism in telecom context.

3.2.3 The SPATEL Engine framework

Having a language sufficiently abstract and expressive for the telecom domain is not sufficient for realizing an effective and agile service development process. Most of the intelligence is to be placed in the model transformers, the code generators and in the execution environments. Immediate execution of service specifications are needed to perform frequent iterations as recommended in almost all agile methodologies. The SPATEL Engine is the native target execution platform for services specified in SPATEL.

3.2.3.1 Architecture of the SPATEL Engine framework

This component provides a default engine for executing compositions specified using the SPATEL language.

The SPATEL Engine framework includes the following facilities:

- A code generator to produce executable code from a service specification in SPATEL language. For elementary services only the stubs are generated. For composite services all the code is generated from the SPATEL definition.
- An internal repository of services to store the code of elementary and generated composite services. Services can be executed using either REST [Fielding00] or SOAP [w3c-soap] protocol, in addition to using HTML forms.
- The generic engine for executing states machines - which is the default formalism for expressing the logic of a composite service.
- Three default interfaces for executing services deployed in the internal repository: a HTTP REST interface, a SOAP interface and finally an HTML/Javascript interface.

In complement of this, additional code generators allow to access services deployed in the internal repository from different execution environments. One example is the code

generator producing simple applications running on a NOKIA S60 smartphone [symbian-s60]. These generators are developed on demand (are not part of the core of the framework).

Model transformations in SPATEL Engine were developed using two techniques: one is through the usage of APIs specific to metamodels, exploiting PyMOF framework, which is a specific implementation of a MOF repository, equivalent of Java-based EMF [eclipse-emf] but for the python language. PyMOF, developed internally in Orange Labs, is a derivation of Universalis work described in [Belaunde99]. The other technique is the usage of QVT operational formalism (see detailed description in Section 2.2.2.3) with SmartQVT tool [smartqvt].

In annex B, Section 9.3 we provide the QVT Operational source code of the SPATEL to WSDL transformation.

3.2.3.2 Variability management

A noticeable characteristic of the SPATEL Engine code generator is that it provides some automatic support to handle variability in the implementation. More precisely, the code generator produces for each service operation three variants in the Python language:

(i) A stub implementation, doing nothing but returning a default empty value (like zero for numeric results, or empty string for string results).

(ii) A "local implementation", launching the state machine corresponding to the behaviour specification (if available), or empty code ready for manual completion (in case of opaque behavior specification).

(iii) A glue code connecting to a declared web-service. The web-service is identified by a key in a registry consisting of a package name, a service name and an operation name. It is up to the developer to update a web service registry to create the link to a real web service.

In annex B, Section 9.4.3, there is an example of generated code for a given service: we can see the Translation::translate() service operation with the three implementation variants.

This capability corresponds in fact to four common cases when developing a service:

- The component to develop already exists as a web service, but we simply need to encapsulate it to make it available as a SPATEL service (known in the internal repository), and possibly adapt its interface. In that case we use the glue variant (iii).
- The component to develop does not exist, but you need to develop it. In that case the developer uses the variant (ii) and performs manual completion.

- The component to develop does not exist, and the designer specifies it completely as a composition of other services using SPATEL state machines. In that case variant (ii) is used and it already contains all the code (no need for further intervention).
- The component to develop is temporarily left not implemented (will be done in an iteration in the future). In that case the *stub* variant (i) is used.

When developing manually a service (variant ii), the service developer benefits from easy-writing and rapid development features characterizing Python applications. Indeed nothing prevents him from developing the functionality in another language and then exploiting web service technology to connect the python implementation to the final implementation.

3.2.3.3 Executing state machines and session management

To the concept of State Machine in SPATEL corresponds a State Machine implemented in Python [python]. Similarly to some VoiceXML [w3c-vxml] systems, the state machine is loaded into memory once at the activation of the service. Then each session object - representing the usage of the service by a user - has a pointer to store its position in the execution of the state machine.

The SPATEL engine relies on an HTTP server to offer multi-threaded and asynchronous support. A session mechanism is explicitly maintained by the framework to allow keeping alive the context when dealing with long running services (containing states waiting for the arrival of asynchronous events).

Two forms of remote execution are supported: one uses CGI protocol, the other uses servlets [jsr-000315] on top of a Java Web Container like Tomcat [tomcat]. In the first case, the HTTP server invokes Python CGI which rebuilds the saved context at each invocation. In the second case a Jython interpreter [jython] is used to connect Java and Python.

3.2.3.4 Construction phase with SPATEL Engine

In Section 3.1.3.3 we presented the proposed agile method for developing telecom services. When using SPATEL Engine, the *construction* phase exploits the available facilities to fulfil each of the 5 tasks. The instantiation of the construction phase with SPATEL Engine tooling is depicted by Figure 17:

Initially, (1) the service designer defines service interfaces in SPATEL with graphical or textual notation or imports service interfaces from WSDL automatically translated to SPATEL. (2) Service interfaces and other entities are translated as class definitions, and state machines are translated into the equivalent behavioral code in the target language (*python* as default). Variability management as described in 3.2.3.2 applies. Opaque operations (i.e. the ones which have only a signature defined but no state-machine available) which are not connected to existing remote services require manual completion. The following optional steps are (3) the immediate execution of the service (for testing) using a fully generated web-based interface, (4) the generation of various widget

applications running on different mobile phones. This may require some manual intervention, except for simple cases where no extra information has to be provided. Finally, (5) if reuse of the composite service is relevant, the new service can be promoted as a new service available as a SOAP web service.

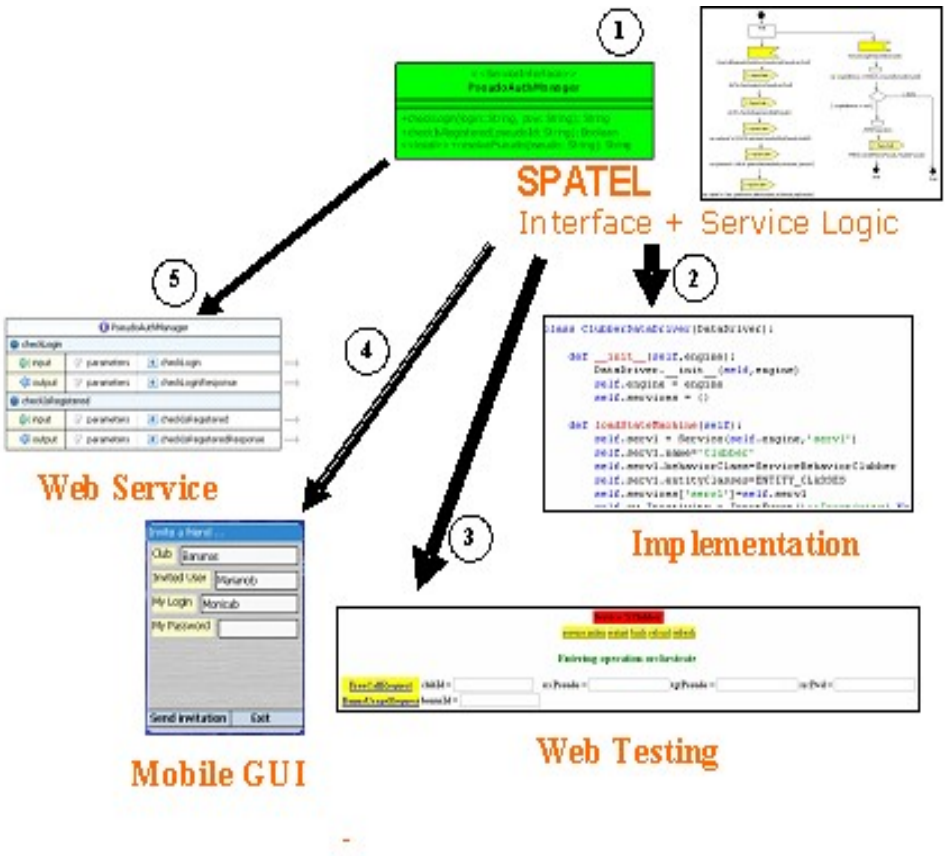


Figure 17: Service Creation Process

3.3 Voice-based Services: Voice DSL and Voice Bench

Interactive voice-based applications are specific telephony applications that are designed to allow end-users to interact with a machine using speech and telephone keys in order to request a service. The interaction – called a dialog – typically consists of a state machine that executes the logic of the conversation and that is capable of invoking business code which stands independently of the user interface mechanism – could be web, batch or speech-based. Because state-machines can be specified and modelled formally, it is possible to design a tool chain that automates large amounts of the dialog implementation.

Notice that the formalism presented here is more specific than the SPATEL formalism in Section 3.2. In SPATEL you may include voice input as a complementary facility in your service definition (see Section 3.2.2.3), but the structure of SPATEL services is not organized in terms of dialogs as it is the case for voice services defined with the Voice DSL..

3.3.1 Voice DSL

In order to serve as a conceptual basis for the voice development environment, a meta-model for platform independent modelling of voice applications was defined and UML 2 was chosen as a concrete syntax. Figure 18 below shows an excerpt of the developed Voice metamodel:

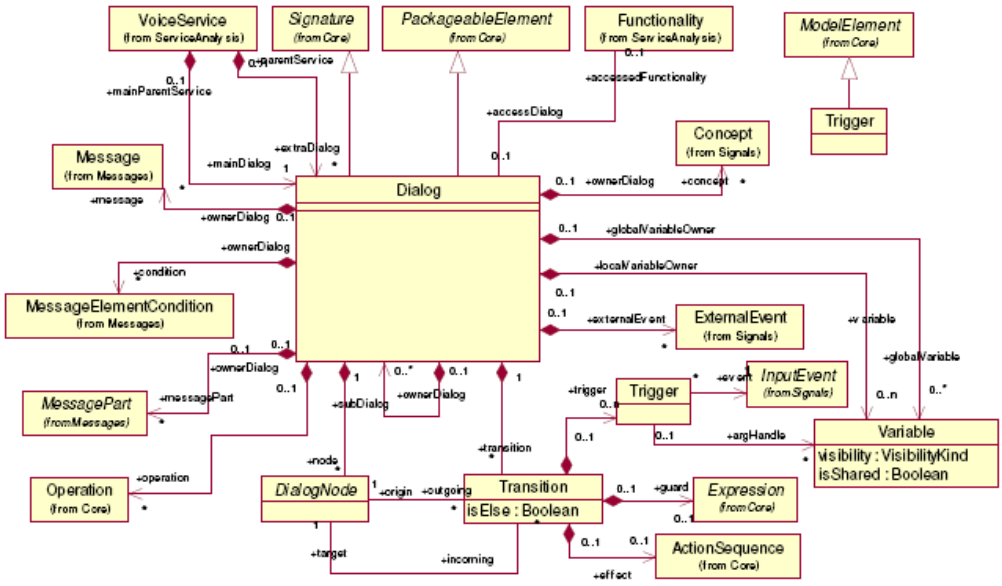


Figure 18: Metamodel for Voice Dialogs

The rationale behind the choice of UML as a concrete notation was:

- Voice application logic can easily be assimilated to a reactive state machine: the application reacts to user input such as voice and telephone keys, and produces output for the user: the vocal messages. The concepts of states and transitions are used in the voice application meta-model and supported by UML.
- Voice applications usually interact with the enterprise's information system. As UML is used as a modelling language in the information system domain, using the same language for voice applications allows seamlessly to integrate information system models with voice application models.

Figure 19 below illustrates the use of UML notation, more precisely a transition-centric state machine similar to the ITU-T SDL [itu-sdl] automate engines:

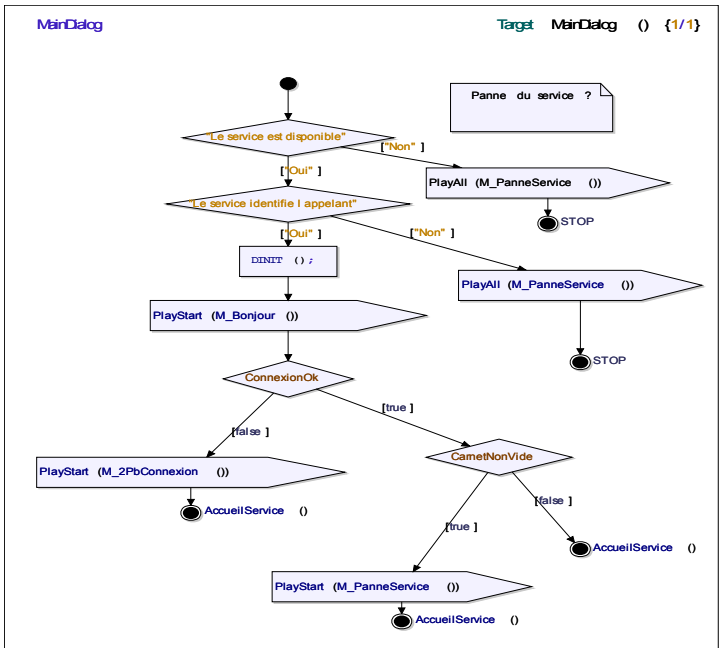


Figure 19: Example of voice dialog behavior

A specific textual syntax was defined as an alternative way to create Voice dialog specifications. Also this syntax was used as a way to interchange between tools (since generating the dedicated text format is often easier than generating the XMI model representation).

A large scale sample of voice service developed with the Voice DSL and associated framework is presented in Validation chapter, in Section 4.2.1 *The address book voice service*.

3.3.2 Voice Bench Tool Chain

On the basis of the voice metamodel, a complete model-driven tool chain was constructed (see Figure 20 below). On the front end there are various alternative UML modelling tools implementing the same Voice UML profile: Telelogic TAU 2.3 [tau] or Objecteering/UML V6 [objecteering] or RSM [rsm], In the middle a model repository to store the voice specifications in terms of the metamodel. On the right the execution engine interpreting a voice specification and a simulator based on IF technology [Bozga04] and associated test generator. The integration between these tools is done thanks to a list of model to model transformers and code generators.

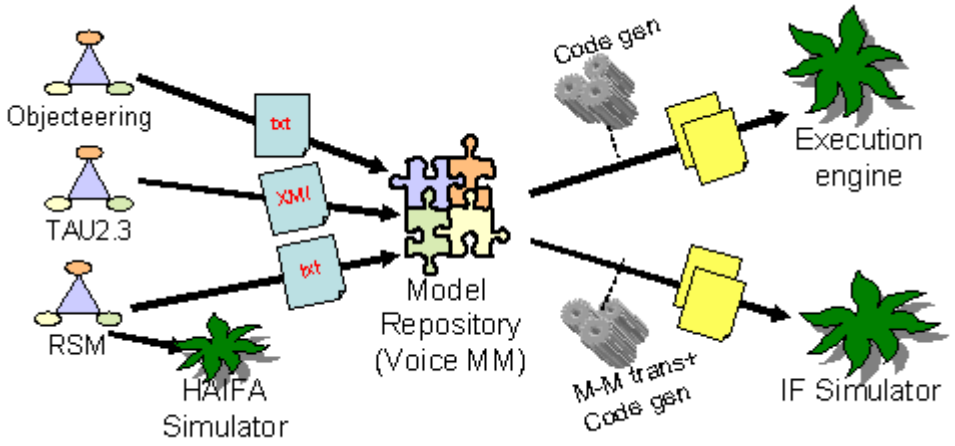


Figure 20: Architecture of the MDD Tool chain

Figure 21 shows the simulation of a voice service. The tool allows seeing the exchange of messages between the different elements:

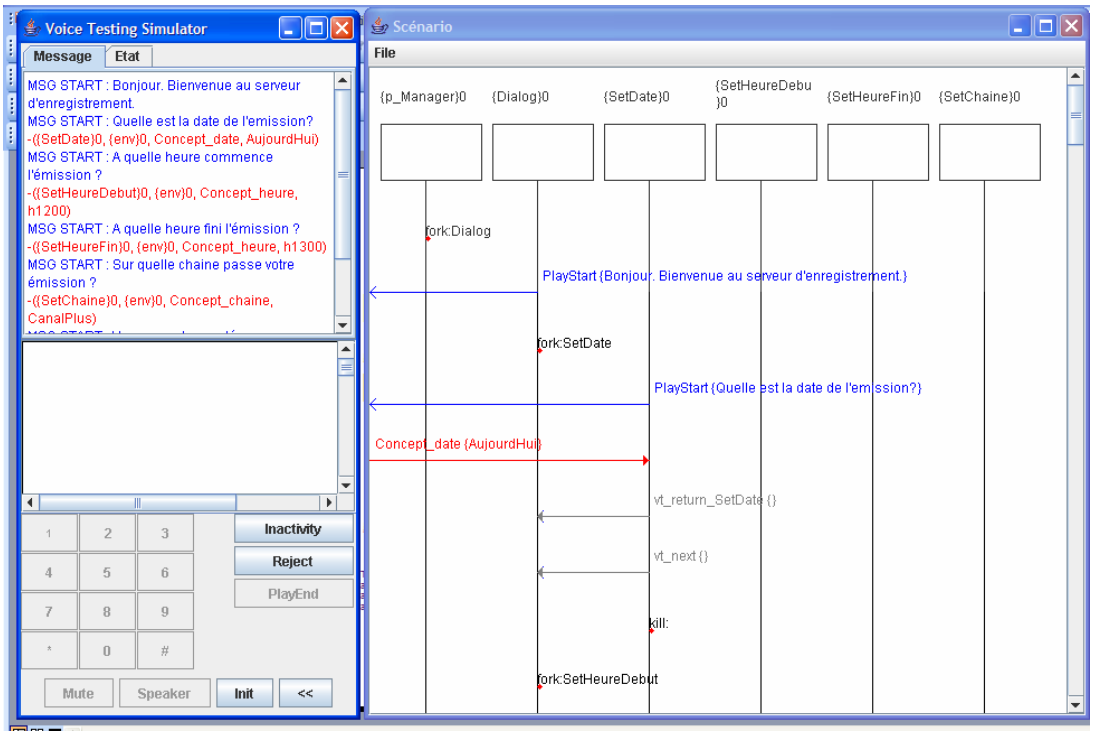


Figure 21: Simulation of a TV Recorder voice interface

An important facility developed within Voice Bench environment was the automatic generation of tests to check correct execution of dialogs. This facility was derived from IF simulation files.

3.4 Contribution Discussion

In this section we first discuss general MDA application issues and their impact on agility. Then, based on our experience on defining and using SPATEL and VOICE formalisms and the associated tooling (SPATEL Engine and VoiceBench), we provide our view regarding the main advantages and limitations of MDA for service development. Finally we state to what extent we have matched the expectations defined at the end of the State of the Art section (see Section 2.5.2 - Criteria of Research).

3.4.1 MDA Application Issues

Despite the potential advantages of a model-driven approach to facilitate and accelerate software development - and this is particularly true for service creation frameworks - we have observed some cases in which an unhappy design decision regarding the choice of an MDA technology can have a significant negative impact on the maintainability of the overall MDA tool chain, and indirectly compromises the agility perceived by end-users of the system (in our case service designers and service developers).

Three relevant examples of sometimes critical decisions to be taken by MDA tool designers are: firstly, the choice between the development of an intermediate model-to-model transformation or the development of a direct model-to-code transformation, secondly, the choice between relying on UML profiles or relying on metamodels to define and support a domain specific language (DSL), thirdly, where to put the border between modelling and coding when dealing with behaviour descriptions.

All three issues will be discussed and then we will provide our recommendations based on our gained experience on applying MDA within France Telecom.

3.4.1.1 Code generation versus model transformation

Ideally code generation should be used only for pretty printing (rendering of a model in one of its possible concrete textual notations), that is, only in the final step of a transformation process whereas model-to-model transformation should be preferred for any process involving a semantic gap between the source and target entities [omg-mda]. Now in practice realizing a model transformation may imply a huge cost when appropriate modelling support for either the source or the target is missing: for instance if someone is in charge of transforming a WSDL service definition into a Java class, he would probably not be happy if he is obliged to define by himself the WSDL and Java metamodels and then spend time on converting standard WSDL files into its metamodel representation (an *injector*), as well as spend time on generating Java code from Java metamodel (a *projector*).

Three problems emerge and are worthy to point out:

- Firstly, the *injectors* and *projectors* may not be available, or impose unacceptable dependencies for being able to use them. As a consequence, the developer will have to develop the injectors and projectors from scratch increasing significantly the overall development cost - not only implementation but also maintenance.

- Secondly, in some cases use of a metamodel representation may bring significant complexity compared to the use of a code generation program exploiting a pre-existing and user-friendly textual syntax. For example a programmer is likely well aware of the Java textual syntax, but may not feel comfortable with the idea of representing it in an abstract manner (in the form of a model conforming to a metamodel for the Java language). In that case thinking in model terms will require more mental effort. Simple operations like a variable assignment that would imply a unique code line in a code generator program may, in the equivalent model transformation specification, imply various transformation rules with a possibly non-trivial sequence of model element creations and update operations. Nevertheless, this last difficulty can be mitigated by the presence of reusable *query* and *constructor* helper operations (like in QVT libraries) that can complement support for a given metamodel. For that reason, in the debate abstract syntax

versus concrete syntax it is not always true that generating a model using the textual syntax is easier than using the metamodel representation.

- Last but not least, we have to take into account some maintenance concerns dealing with metamodel based representation: what happens if changes are needed in the metamodel? How does this impact the previously written transformations? Metamodels complexity tends to grow fast with language size (typically having a complex hierarchy of concepts) while textual syntaxes are generally less sensitive to enhancements. A refactoring of the class hierarchy can be needed for an enhanced metamodel while the equivalent in the textual form can be done by simply adding a non terminal keyword.

Our recommendation

From the point of view of software development agility, the choice between using a code generation technique against model-to-model transformation should be considered seriously and should as much as possible be driven by pragmatics. The kind of questions to be raised before making a choice is:

- Do I have a pre-existing support for source/target metamodels that I can reuse without significant effort?
- Do I have an alternative textual syntax for which tools are already available?
- How stable is the metamodel?

3.4.1.2 Meta-modelling versus UML profiles in service modelling

In Section 2.2.2.4, Domain Modeling, we introduced the *Metamodel versus UML Profile* debate. We provide here our own analysis and recommendation on the basis of our experience with the SPATEL and VOICE DSLs.

Beyond the problem of "meta-modelling versus UML profiles" we have two distinct questions:

- What is the better technique to define the abstract syntax of a specific domain?
- What graphical concrete notation should I use for my domain?

There is an ambiguity regarding usage of UML Profiles: the motivation for using it may be reuse of UML concepts or reuse of UML graphical notation or, more often, a mix between the two. Unfortunately, in the UML Profile mechanism, notation customization is, from our point of view, not sufficiently flexible to avoid pollution of concepts with notation concerns.

For instance we have the following two restrictions that apply to stereotypes: (i) the display name of a stereotype is the name of the concept, and (ii) a stereotype specializes a unique concept.

As a consequence of (i), the profile designer may favour imprecise terms that are fine from a notational point of view but not from a conceptual point of view. An example is the <<*service*>> stereotype in SoaML [omg-soaml] used to denote *service access points*.
As a consequence of (ii), in order for a domain concept to have multiple representations (like the Activity concept represented as a UseCase and as an ActionNode in SPEM 1.0 [omg-spem]) the profile designer will be obliged to define various stereotypes to represent the same domain concept. Hence, conceptually there is a mismatch between the list of stereotypes and the list of domain concepts in the profile.

Another difficult issue with UML Profiles is how to exclude UML concepts and properties that have no meaning in the domain. Use of OCL [omg-ocl] for this purpose is indeed possible but demands an intensive effort. Moreover, due to the big size of UML, it is difficult to assert that all undesired cases have been treated.

Now, regarding the second question, UML Profiles may be in competition with ad-hoc graphical notations that can could be implemented through non-UML meta-case tools. An example of a meta-case tool is MetaEdit+ [metaedit] that allows attaching declaratively graphical elements to metaclasses.

Nevertheless, usage of UML based notations for domain specific purposes have some important advantages:

- Availability of the UML tools at relatively low costs,
- Reuse of stable and standardised notation, which means less cost to learn and understand it,
- Enhanced Profile support in some UML tools to allow a high degree of customisation (like hiding of unused diagrams).

Our recommendation

Taking into account the previous discussion, our position regarding the first question (better technique for abstract syntax definition) is to favour metamodel-based representation to define domain concepts (that's why the SPATEL and VOICE DSLs are primarily defined by metamodels). Serialization and transformations can then be achieved on the basis of a clean formalization of the domain. Regarding the second question (what graphical notation to select) our position is to try to use as far as possible UML based notation (hence SPATEL and VOICE use UML as concrete graphical syntax).

Such complementary usage of metamodels and UML modelling has however the problem that the tooling needs to handle two different representations, and hence requires to maintain their consistency through transformations.

3.4.1.3 Graphical modeling versus coding of service logic

The target users of the SPATEL or VOICE graphical notation are professional service architects and service developers. The first population of users will probably not have to deal with the implementation tasks. However, for the second category of users, we can legitimately ask whether it makes sense to develop the logic of a service using a graphical notation instead of using directly a general purpose programming language.

Our experiments leads us to the observation that, for sure, for a programmer, using a graphical notation is much more time expensive than direct coding. However, if the time for providing an implementation is not a critical issue, there are clear advantages to make use of a graphical notation to develop service logic that has good quality:

- Firstly, in formalisms like SPATEL, the designer is free to decide where to put the border between "graphical design" and "textual coding" of service logic: any intensive computation can be encapsulated by means of a black-box local operation. Also, some components may be completely implemented using opaque code and still have a well-defined SPATEL interface to allow its reference in other services. This emphasizes the fact that the choice between graphics and text is not black or white. The good balance between both is the responsibility of the service writer.

- Use of graphical notation helps clarify ideas ("What is conceived well is expressed clearly") and hence to define service logic that can be understood and validated by others. Quality of abstraction is an important feature for those who want to apply model-driven transformations to create multiple implementations from the same specification.

We believe the problem of the border between design and code will always exist. However we notice that model-driven technology is effectively pushing in the direction of making more and more design and less coding and this is particularly true in the domain of service development.

Our recommendation

A good DSL for service development needs to offer the *flexibility* that allows the user to decide whether to make behavior explicit or to leave it opaque at modeling level. This is typically provided by "black-box" operations or through "informal actions or conditions".

3.4.2 MDA advantages for service development

In this section we point out some observed benefits of using model oriented engineering to develop telecom services.

3.4.2.1 Enabling vertical and horizontal variability

MDA applied to service development allows two kinds of variability, vertical and horizontal.

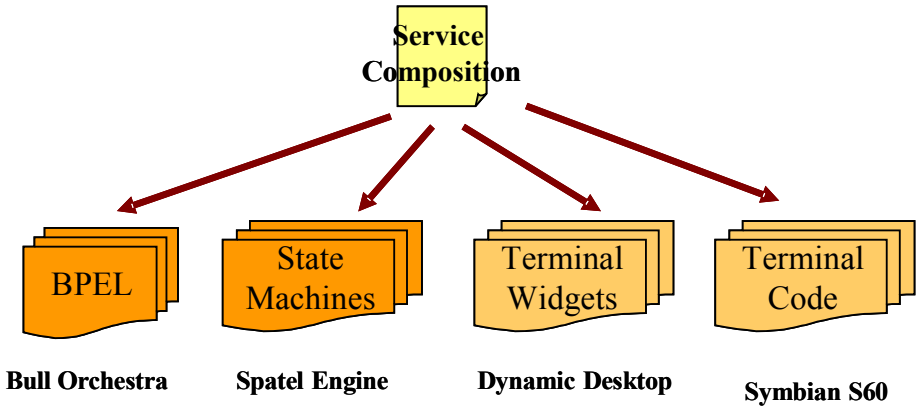


Figure 22: Vertical Variability

Vertical variability is the ability to run a specified service logic in potentially various execution platforms. Figure 22 depicts the deployment of service logic in an instance of the SPATEL engine as well as on top of a BPEL engine, or even part in smartphones terminals. A second kind of variability, which we call *horizontal variability,* allows replacing an invoked component by another, by simply adapting the implementation on the basis of a neutral common interface.

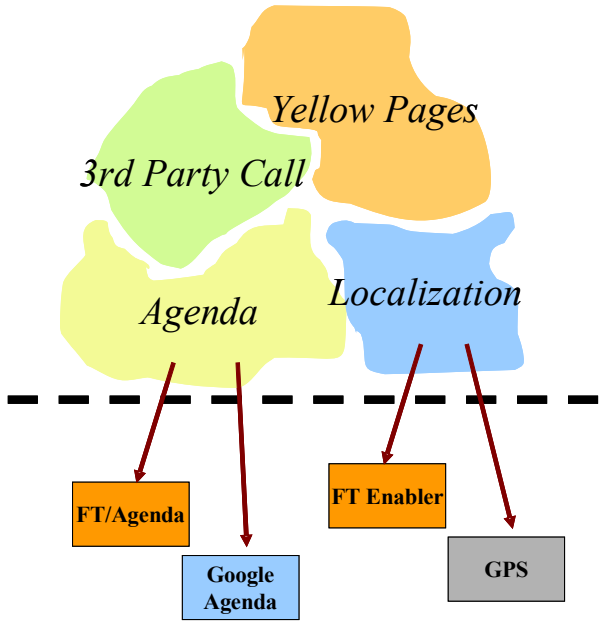


Figure 23: Horizontal Variability

In Figure 23, the click-to-call component used to provide a phone call facility (see illustration scenario in Section 4.1.2) has two alternative implementations that can be called by the composite logic: for instance an implementation based on the Asterisk platform [asterisk] or a dedicated enabler offered by Orange. As described in Section 3.4.2.1, to facilitate horizontal flexibility a framework, like SPATEL Engine, produces various implementation variants for each service operation. Efficiency of vertical and horizontal variability depends a lot on the characteristics of the software developed to support the DSLs.

3.4.2.2 Inserting non-functional behaviour thanks to code generation

One advantage of code generation applied to the development of composite services is the ability to transparently insert some behaviour before of after a service call. For instance, depending on security configuration data associated to a service, SPATEL Engine generates control code previous to the invocation of a service to check that the user is permitted to call a service. Other kinds of non-functional concerns can be added, like generating events to a monitor system to control the execution.

Conceptually non-functional configuration of services behaves as aspects that influence code generation (see reference to Aspect Oriented Modeling in Section 2.1.1).

3.4.2.3 Tool interoperability

The VoiceBench framework to support voice service development (see Section 3.3.2) was built on the basis of pre-existing commercial tools to offer essential capabilities, like model editing (Telelogic TAU and IBM RSM), simulation (IF Engine) and testing. The

connection between these tools was realized thanks to metamodels - the VOICE metamodel serving as pivot representation of various modelling tool - and thanks to transformations (such as the generation of IF code). A lesson learned is that inter-operability between tools that were originally not designed to work together can be enabled by the exploitation of MDA technology.

3.4.3 MDA limitations for service development

In this section we point out some observed limitations or drawbacks of using model oriented engineering.

3.4.3.1 Cost of changing the DSL metamodel

The efficiency of a DSL like SPATEL and VOICE highly depends on the maturity of the accompanying tool, like the availability of model checkers and the availability of various transformers to ensure service executability in various popular platforms. However, during the project is it sometimes the case that non trivial enhancements to the metamodel on which all utilities are based need to be done to take into account new user requirements (in the case of SPATEL, there were two major revision of the metamodel). In fact, unsurprisingly we observed that the cost of such changes was high, due the number of utilities already implemented.

This is probably one of the main drawbacks of metamodels: the more it is used and supported, the more difficult it is to insert changes. To mitigate this risk, metamodel adaptation techniques can be put in place, like those that are specified in *QVT modeltypes* or by means of utility libraries that encapsulate the access to the metamodel.

3.4.3.2 DSL learning curve

An obvious drawback of inventing new DSLs is the learning curve for acquiring appropriate skills to use it. To minimize the risk, we have tried to follow an expression and instruction syntax similar to the JavaScript for the textual notation and for the graphical notation our preference was reuse of UML.

Independently of the characteristics of the DSL notation, tooling facilities are essential to facilitate learning of language, such as availability of editors with colouring and completion facilities.

3.4.4 Summary of contribution

In Section 3.1 we presented our approach for achieving agility when developing telecom services. In Section 3.2 we described in detail our proposed solution in the case of integrated composite services and in Section 3.3 in the case of interactive voice services.

Our research work has focused on the two aspects identified in State of the Art part as being not yet well studied (see Section 2.5.2), which are an *integrated* formalism to designing telecom services and the development of a *model-aware* service creation and execution environment. The SPATEL formalism (Section 3.2.2) integrates various aspects of service design including dynamic behaviour, semantic description, non-functional features and minimal user interface definition to derive graphical or vocal interaction interfaces. On the other hand, the SPATEL Engine and the VoiceBench frameworks provide necessary machinery to speed-up the development process (immediate simulation, automatized deployment, test generation) and to simplify maintenance and evolution - like support of *implementation variability* in the SPATEL Engine (Section 3.2.3.2).

In our contribution we have also emphasized the necessity to apply MDA with pragmatics in order to avoid some risks regarding inherent MDA complexity (see Section 3.41 and 3.4.2).

4 Chapter - Validation

This section focuses on the validation of our contribution on the basis of a series of experiments and an evaluation of productivity gain obtained with the MDA tooling.

Firstly we provide an overview of the validation method with objectives of each experiment and hypotheses to verify. After that we describe each experiment in detail with individual evaluation. Finally we provide a summary of our conclusions regarding the hypotheses.

4.1 Validation Overview

What are the benefits in terms of agility implied by the development and usage of the model-driven tool chain for service creation and what are their costs?

These are typical questions that we have to answer to evaluate the return of investments of using MDA for service development. In this section we will attempt to state some conclusions in relation with some hypotheses.

The three hypotheses we want to verify are:

H1: *Use of MDA in service creation tools enhances productivity of service designers and service developers.*
H2: *Explicit modelling of service logic facilitates service evolution, even for complex services*.
H3: *Service creation tools exploiting MDA are difficult to develop but easy to maintain*.

We should point out that H1 and H2 concern users of a service creation environment (service designers and service developers) whereas H3 concern developers in charge of creating the model-driven service creation environment.

In the next section we present three experiments and their evaluation in relation with the selected three hypotheses.

The first experiment concerns the development of a large voice service (a service to access an online address book using voice) achieved in two ways: with a "traditional approach" and then with model-driven technology. This experiment will help us to evaluate H1. The following experiment concerns the development of composite services combining telecom and IT resources: a dinner planning service for tourists. These two experiments will help us to evaluate H2 and indirectly H1. The third experiment concerns the effort for developing and maintaining a MDA tool chain - in fact the VoiceBench tool chain introduced in Section 3.3.2. This will provide inputs for hypothesis H3.

The last sub-section in this validation chapter gives our conclusions and additional feedback (lessons learned).

The following section contains the evaluation of the experiments. For the first experiment a detailed *quantitative* evaluation of the productivity gain has been done. For the two other experiments the evaluation is more qualitative and focuses on gained flexibility.

4.2 Experiments

For each experiment we provide the scenario definition, some information how it was implemented (experiment realization) and finally an evaluation in relation with one of the hypotheses we want to verify. Evaluation of experiments 1 and 4 are based on quantitative measurements whereas for experiments 2 and 3 the evaluation is done qualitatively.

Note: Annex C reports on an additional experiment, concerning creation of simple composite services using *natural language*. It is not presented in this chapter because it is not finalized yet.

4.2.1 Address book voice service

4.2.1.1 Objective of the experiment

The Address Book voice service allows users to consult entries in their address book and trigger calls and to realize some simple editing operations using voice. This service was developed firstly using a "traditional approach", which includes a precise specification step using an ad-hoc formalism, which serves as documentation to implementers. The same service was then implemented using VOICE DSL with a UML tool and then with the

facilities provided by the VoiceBench MDA tool chain. The objective of this experiment is to test validity of hypothesis:

H1: *Use of MDA in service creation tools enhances productivity of service designers and service developers*

4.2.1.2 Description of the service

The main features of the address book service are:

- Ability to consult the contents of the address book;
- Ability to ask to call someone on the basis of a "name" or a phone number pronounced by the user;
- Ability to add entries in the address book.

The following 5 modules are defined in relation to the mentioned features. Calling a contact implies the identification of a contact name or the identification of a telephone number.

Consult Contact Module

This module defines the interaction for consulting an address book. The end-user of the service can ask for the details of one specific entry or may ask to listen to all the entries of the address book. In the latter case, the end-user can interrupt the machine when the person sought is pronounced. After that he can typically activate the other modules to call the contact (Set up communication module) or to update the contact record (Update module).

Identify Contact Module

This module defines the interaction to retrieve the phone information concerning a contact that is in the address book of the user. Either a nick name or the official name of the contact is pronounced. The dialog needs to manage possible duplicates by asking the user to disambiguate.

Identify Number Module

This module defines the interaction to retrieve the contact information concerning a phone number: it tries to find the contact record of the user and reports to the user on its search result. This module is used to know whether it is useful to ask the user to add a contact, when the end-user receives a call of an unknown person.

Set up communication Module

This module defines the interaction to establish a phone call with a contact of the address book. The end-user may pronounce the name or nick name of a contact, then he may indicate the phone on which the callee will be contacted (mobile, fix phone, and so

on). After the communication ends, it is possible to resume the interaction by sending a star DTMF input.

Update Module

This module defines the interaction to update an address book contact. The end-user of the service may ask for an update directly by pronouncing the contact name or by context, i.e., after a conversation involving the contact person. The fields that can be updated are: the name, the nick-name, and the phone number with its category (mobile, fix) and its usage (professional, home).

4.2.1.3 Realization

The realization indeed was quite different between the non model-driven approach and the model-driven approach. In annex A we provide information on the realization using the traditional approach. The following sub-sections concern the realization using the MDA tooling.

4.2.1.3.1 Design highlights

We provide here some highlights of the development of the service by showing some of the artefacts of the design and implementation phase. Figure 24 shows the main dialog interaction.

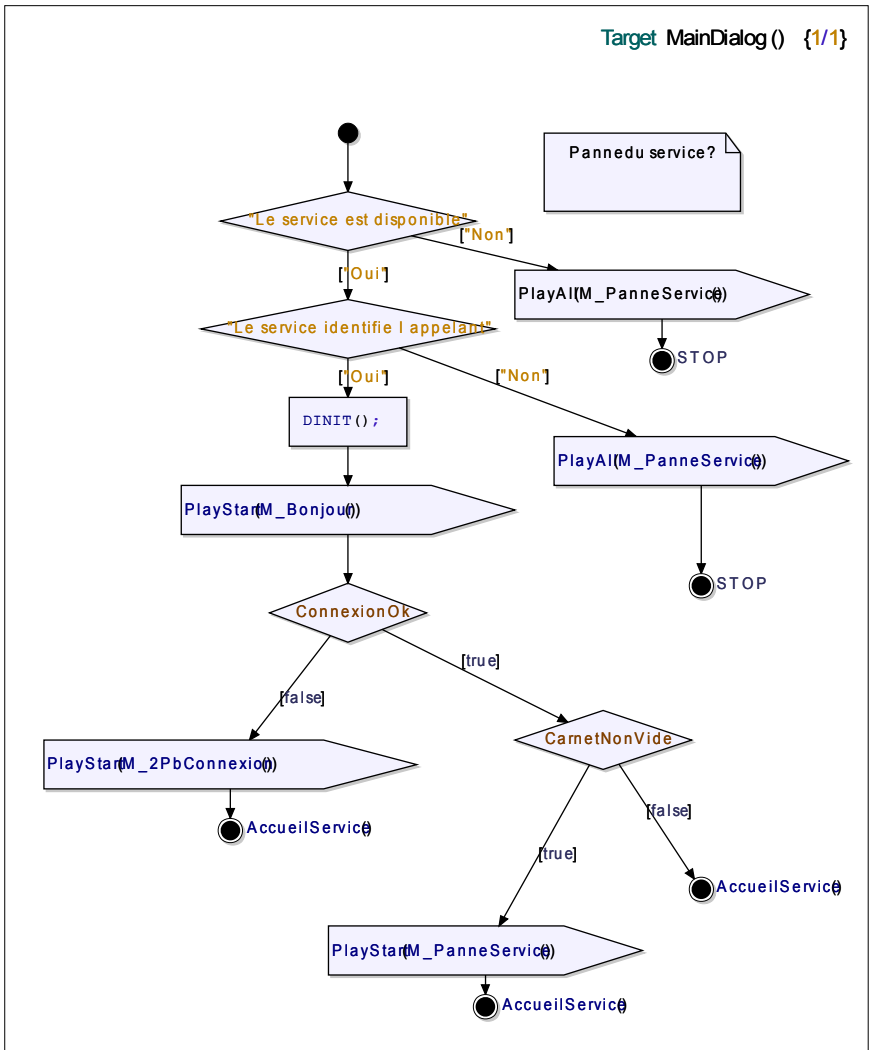


Figure 24: Main Dialog of Address Book Service

Firstly the system checks if the service is available and whether the user is authorized to use the service. Then the initialization dialog is entered (for variable initialization). Then the dialog checks the connection and the existence of the address book (decision node "CarnetNonVide" in the diagram). Then a redirection to the "home" dialog of the service is done.

Figure 25 shows the sub-dialog that is in charge of capturing from the user the minimal contact information required to search the address book and eventually make the call. The contact information can either consist of the first name, the last name or a combination of the two. An input symbol from the "Wait" node represents this event expectation.

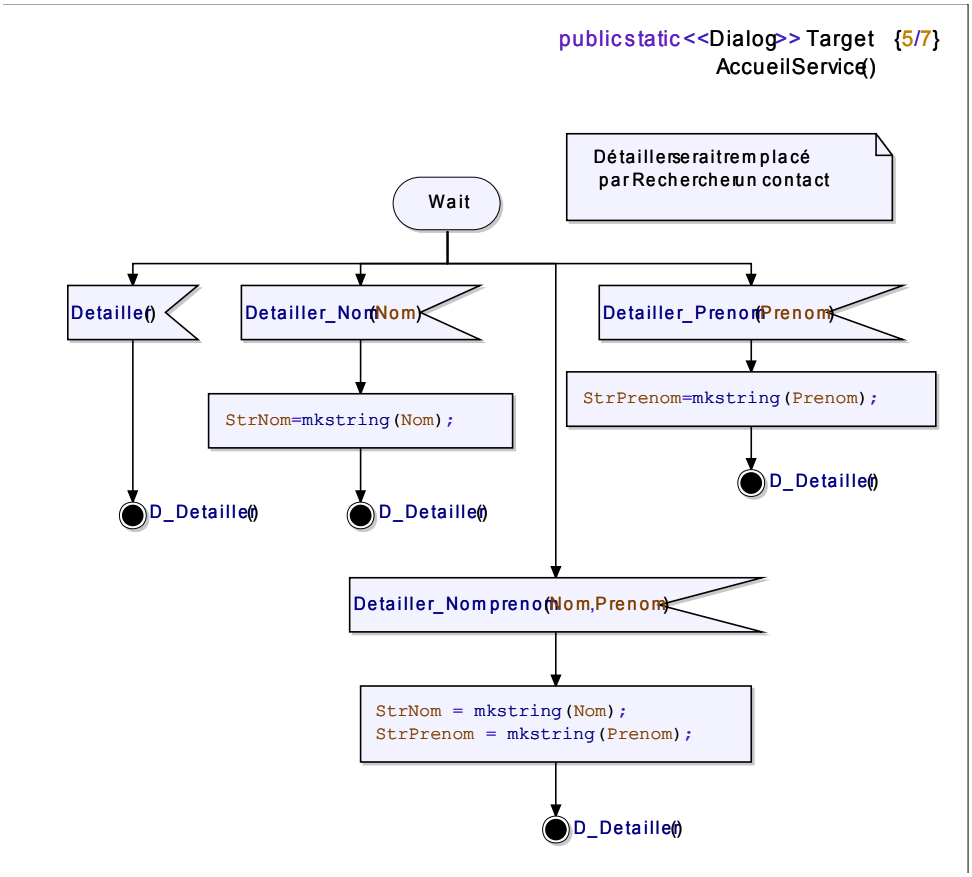
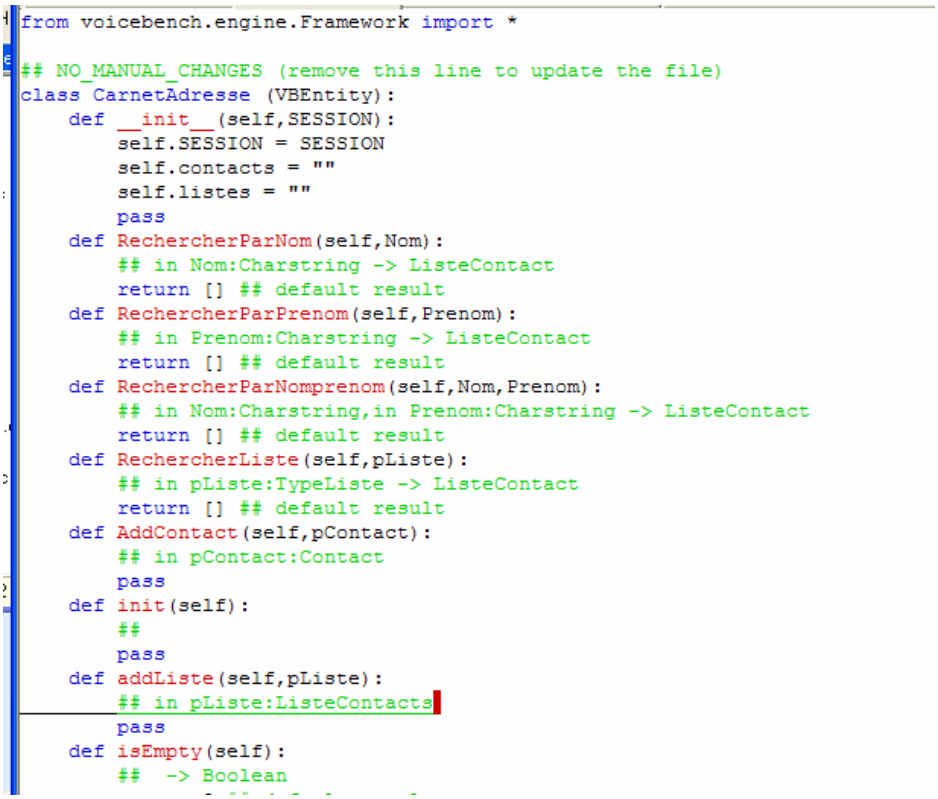


Figure 25: Dialog to retrieve contact information

The model specifies the minimal interface to access the address book component but the actual implementation of the address component is done outside. The task of the service implementer consists of linking the generated code to the existing address book software component. The code example below shows the generated Python code for the Address Book external entity (class CarnetAddress in French). We see here the list of declared operations and the default code generated which allows us to simulate the service even if the connection to the actual implementation code is not done. The presence of the "NO MANUAL CHANGES" line is an indication that the file is actually not yet manually edited. If a manual edition is done, the developer has to remove this line to ensure that a new generation preserves the file.



```
from voicebench.engine.Framework import *

## NO_MANUAL_CHANGES (remove this line to update the file)
class CarnetAdresse (VBEntity):
    def __init__(self,SESSION):
        self.SESSION = SESSION
        self.contacts = ""
        self.listes = ""
        pass
    def RechercherParNom(self,Nom):
        ## in Nom:Charstring -> ListeContact
        return [] ## default result
    def RechercherParPrenom(self,Prenom):
        ## in Prenom:Charstring -> ListeContact
        return [] ## default result
    def RechercherParNomprenom(self,Nom,Prenom):
        ## in Nom:Charstring,in Prenom:Charstring -> ListeContact
        return [] ## default result
    def RechercherListe(self,pListe):
        ## in pListe:TypeListe -> ListeContact
        return [] ## default result
    def AddContact(self,pContact):
        ## in pContact:Contact
        pass
    def init(self):
        ##
        pass
    def addListe(self,pListe):
        ## in pListe:ListeContacts
        pass
    def isEmpty(self):
        ##  -> Boolean
```

Figure 26: Generated code for the Address Book Entity

4.2.1.3.2 Simulation and Execution

The code below shows an excerpt of IF simulation state machine generated for the previous sub-dialog. This code represents a translation of the state machine originally provided in UML form with some specificity for testing.

```
state Wait;
  input Reject();
    task state_history := 2;
    nextstate SubDialogState_D_Reject_Inactivity_1;
  input Inactivity();
    task state_history := 2;
    nextstate SubDialogState_D_Reject_Inactivity_2;
...
  input Concept_Composer_numero();
    skip;
    nextstate DiversionNode_D_Composer_3;
  input Concept_Arreter();
    skip;
    nextstate act_Wait_7_0;
...
  input Concept_Detailler_Nom(({p_Manager}0).vt_context.Nom);
    skip;
```

```
    nextstate act_Wait_16_0;
...
endstate;
```

In order to be executed by the VoiceBench execution engine the service description is compiled into a list of Python files. Then the service can be deployed and immediately executed using a web-based interface, prior to call the service through voice.

Figure 27 below shows a screen-shot of the web-based execution of the service.

Figure 27: Immediate web execution of the Address Book voice service

To each interaction web page corresponds a VoiceXML page, which is used when the access is done through voice using a phone.

4.2.1.4 Evaluation and lessons learned

4.2.1.4.1 Hypothesis and threats to validation

The hypothesis to be verified is:

H1: Use of MDA in service creation tools enhances productivity of service designers and service developers.

The following *threats to validity* need to be considered:

- Number of tested services: The voice service implemented in the experiment using the two approaches (traditional versus model-driven) is a large one and has typical complexity of this kind of services. However, it remains that it is only one unique example for which we were able to provide comparative measurements.

- Tool maturity: Development of the service using MDD suffered from the immaturity of the developed tool (perfectible ergonomics, code generation bugs, and so on).

4.2.1.4.2 Evaluation Summary

Annex A in Section 8.2 presents the detailed quantitative evaluation. We provide here the summary of the outcomes of this study.

The study provides an interesting indication on the productivity change that can be obtained when using the MDD tool chain for developing voice applications. For the design phase, we obtain approximately 20% of productivity gain, whereas for implementation activities we obtain a very high rate of 70%. This is easily explained by the following reasons:

- For the design phase, the increase of productivity obtained thanks to the use of the modelling tool – in contrast with the usage of MS word tables – is mitigated by the fact that the service design has to spent some significant additional intellectual effort to build a specification that is complete and non ambiguous. In effect, usage of the tool, including the simulation capabilities, enforces the quality of the model to be at an acceptable level.
- For the implementation phase, since a large part of the implementation is generated, the time spent on providing the glue code to connect to the business entities is dramatically reduced.

Of course, this observed productivity gain does not take into account the cost of the development of tool chain (this is evaluated in Section 4.2.4).

To conclude, based on the quantitative study, we consider hypothesis H1 verified, at least for the specific domain of this study, which is voice service development.

4.2.2 Dinner planning composite service

4.2.2.1 Objective of the experiment

The experiment described here concerns the development of a composite service integrating telecom and IT facilities. For the service provider it may be important to be able to replace one component by another to take into account changes in his environment - like a strategic partnership change (like moving from Google Calendar to Orange Calendar). We will use this experiment to verify hypothesis:

H2: *Explicit modelling of service logic facilitates service evolution, even for complex services*.

4.2.2.2 Service Description

The E-tourism dinner planning scenario is as follow:

- An End User is on travel in a city. Because he does not want to waste time trying to find a good restaurant for his dinner he will delegate this task to a specialized dinner planning service. In the morning, he sends an SMS to the Service dinner planning requesting for finding a "recommended" restaurant at 20:00 nearthe location where he will be at that time, and respecting some criteria (type of food),

- At dinner time (20:00), the Service locates suitable restaurants based on the end user geographic position,

- The Service sends a message to the End User containing the list of restaurants located in the surroundings including the contact points for reservation,

- The End User activates a call to the restaurant of choice using the restaurant contact point information.

The components that need to be in place for this scenario are:

- A Personal Agenda, to store from the user his willingness to be notified at dinner time,

- A Localization service, which will find the user's location relying on GSM network information,

- A SMS or Instant Messaging enabler to notify the user when the list of restaurants is found,

- A Yellow Pages service to find the restaurants near the location of the user,

- A Third Party Call component to activate the call to the selected restaurant.

Figure 28 below shows the interaction between the different composed components and the orchestration engine:

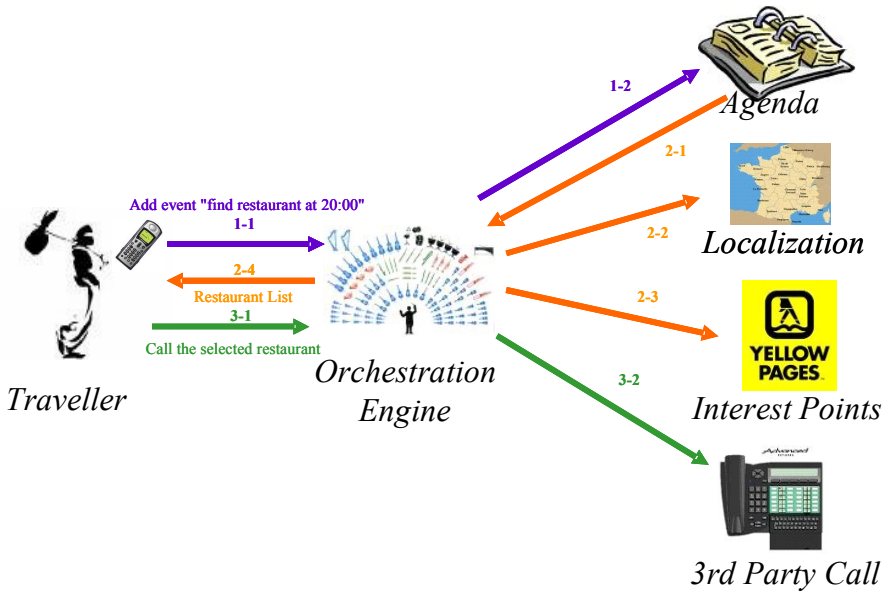


Figure 28: Dinner planning scenario overview

From the point of view of the orchestrator, the scenario has three temporal phases:

- The orchestration engine receives the user request (1-1) and registers the event in the personal agenda (1-2),
- At dinner time, the orchestrator receives the reminder from the personal agenda (2.1) and subsequently invokes the localization services (2.2) to obtain the location information of the traveler. Then it requests the interest points of the yellow pages services (2.3), collects the responses and sends the results to the traveler (2.4).
- Finally, if the user selects a restaurant, the orchestrator receives the request (3.1) and invokes the 3rd party call service to establish the communication.

4.2.2.3 Realization

4.2.2.3.1 Design of the composite service

In our experiment, the SPATEL language, described in Section 3.2.2, has been used to develop the dinner planning service. In practice, following the SPATEL language philosophy, this means:

- Declaring the interfaces for all the invoked components (Agenda, Localization, Yellow Pages, 3rd Party Call),
- Declaring the composite component – with a single 'orchestrate' operation – and defining the logic of this operation through a state machine.

All of the components to invoke already exist in some form. The Localization component is provided by Orange in the form of a web service, the Interest Points restaurant inspection can be obtained using an HTTP GET request on the French "Pages

Jaunes" web site (after some filtering and parsing of the HTML output), the 3rd Party Call is another web service, and the agenda on line web component role can alternatively be played by Google Calendar application or a specific Orange Personal Calendar service.

So at this level, various questions arise, like:

- When a web service, is available should I directly derive the SPATEL interface from the WSDL interface or should I try to make some filtering to simplify it?
- When we have more than one candidate, should I try to define an interface that works for all the available possibilities?

Taking the WSDL file "as is" – through the WSDL to SPATEL importer – could be a comfortable solution but has some drawbacks. For instance, it could have an impact on the complexity of the service logic definition, due to the fact that additional parameters - not really relevant to the designed composite service - may need to be constructed and passed anyway to have a valid service invocation.

Concerning the second issue, abstracting a common interface implies that there is the possibility to make the adaptation somewhere – maybe at deployment, when generating code from the model of the logic, or, at runtime, when executing the service through an intermediate object that performs the argument conversion. The best choice really depends on the target execution technology. When using the BPEL engine we tend to favour the first solution relying on code generator intelligence to perform the interface adaptation, since adding an intermediate web service would be costly. In the case of the SPATEL Engine, for which an intermediate local proxy class is always generated, the second solution is much more convenient.

In the case of the Dinner Planning service we followed the strategy of abstracting and simplifying as much as possible the interfaces of the invoked services. In the end, this had some implications regarding the design of the Service Repository: a unique SOAP web service may be associated to one or more registered SPATEL interfaces.

Figure 29 shows the interface of the Agenda component which abstracts a piece of functionality common to the Google Calendar and the Orange Personal Agenda component.

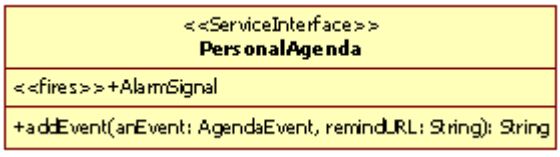


Figure 29: Interface of the Personal Agenda component

Figure 30 shows the modeling of the logic of the orchestration operation: we see the three threads of execution.

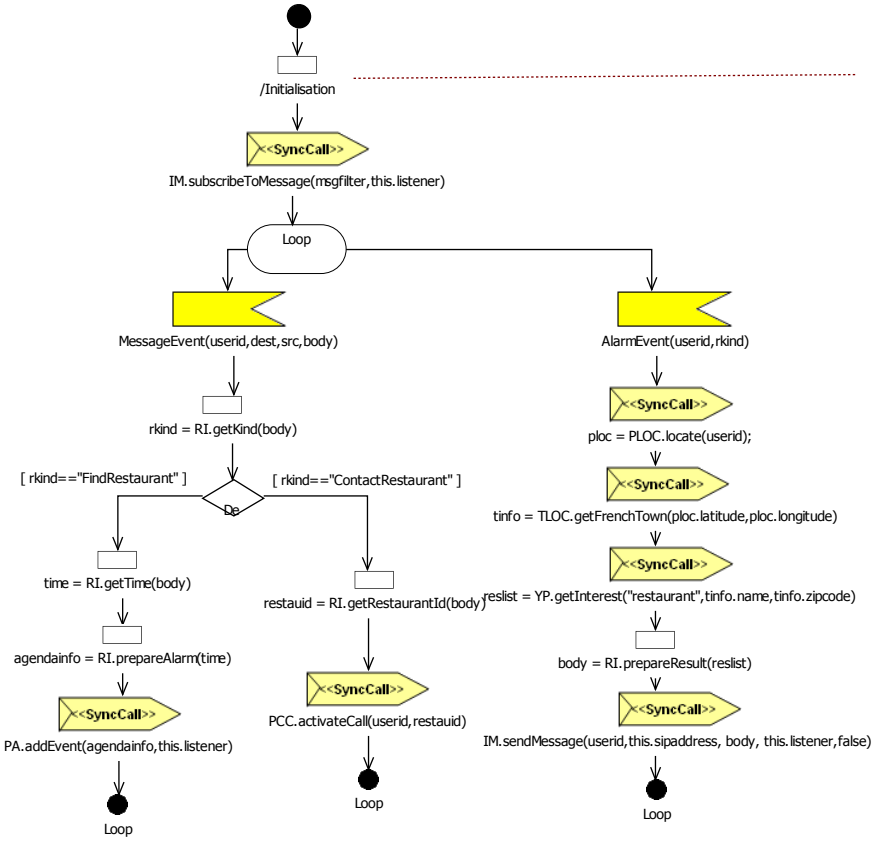


Figure 30: Logic of the dinner planning service orchestration

On the left, we have the reception of the user initial request, on the right the treatment of the event triggered at dinner time and in the middle the final phone call. Note that this state machine uses the new UML2 transition centric view - in fact taken from ITU SDL – in which the the actions executed during the triggering of a transition are explicitly represented as rectangles. In this diagram a specific icon is used to denote a remote service invocation, similar to an asynchronous signal sending symbol in UML. For the comprehension of this diagram, we should also mention that a Service Call in the SPATEL formalism is not an action but a State node, which gives the possibility for defining explicit exception transitions in case of invocation errors - overriding the default mechanism for handling errors.

4.2.2.3.2 Implementation and deployment of the composite service

We generate two alternative implementations: one on top of the BPEL engine and the other on top of the SPATEL engine. In our development process, the implementation is the engineering phase where code generators are invoked and code completion is done when necessary. Because the state machines used in SPATEL have unambiguous execution semantics, the code corresponding to the state machine was completely generated. The part that required some manual code completion was the code related to the realization of "non

standard" remote service operation calls, like the one performed to connect to Google Calendar [gcalendar] since this follows a proprietary protocol. Also all intermediate computations – like the formatting of the message containing the list of restaurants, which were modeled as invocations of local black-box operation calls – need to be completed, since only the skeletons were generated. The percentage of generated code in our dinner planning application was 80%. Notice however that in situations where all invoked components represent already existing components - registered as implemented components in the SPICE service repository - this generation factor may be 100%. The richer is the catalogue of services, better are the chances to produce composite services without any code writing.

The client part for the Nokia N80 phone was generated using a specific transformer exploiting a description of the GUI elements in SPATEL (see GUI support in Section 3.2.2.4). Later on we also produced a widget interface for a windows mobile phone using the Dynamic Desktop Mobile framework from Alcatel [spice-d83]. Figure 31 represents the screen to activate the service in a Nokia phone.

Figure 31: Activation menu for the dinner planning service

4.2.2.4 Evaluation and lessons learned

4.2.2.4.1 Hypothesis and threads to validity

The hypothesis to be verified is.

H2: *Explicit modelling of service logic facilitates service evolution, even for complex services*.

The following *threat to validity* needs to be considered:

- The developers of the *Dinner Planning* service were essentially the developers of the orchestration tooling. Detailed knowledge on the flexibility brought by the variability mechanism in SPATEL Engine may have influenced the way the service was designed and hence facilitate in the end component substitution.

4.2.2.4.2 Evaluation Summary

There can be different kinds of service evolution. In our example, service designers may for instance (i) enrich the service logic changing the specified composition algorithm - possible invoking additional services. It may also (ii) replace one of the invoked components by an equivalent one. Finally it may (iii) add some non-functional behaviour behind the scenes - like adding some access control before actually invoking a service.

In our example, the first case (i) implies changing the SPATEL algorithm. The user benefits from automatic code generation to re-apply service simulation and execution on the modified logic. The second case (i) may not imply any change in the logic, if the replacement component satisfies the same interface. Changes are typically done in configuration files and in the generated stubs to connect to the new service. The third case (iii) could be provided directly by the framework (SPATEL Engine) without any need of change.

The dinner planning service experiment has demonstrated the ability to easily replace components (Google Calendar replaced by Orange Personal Calendar), especially when the reference interface is defined in an abstract manner to avoid proprietary dependencies (like the PersonalAgenda interface depicted in Figure 29). The replacement effort consists mainly of a simple code realizing the interface adaptation code (less than 10 lines of Python code in our case).

In contrast, such service evolution is more difficult to realize if the whole service logic is implemented directly by programming code and if the written code exposes low-level decisions - like creating SOAP messages to invoke a web service. In SPATEL, when we are expressing a service invocation, we do not know what protocol is used.

4.2.3 Development of a MDD Tool Chain

4.2.3.1 Objective of the experiment

To assess agility of applying MDA in service development we should not only consider the agility perceived by a final user of the tool chain when the development of the tool is completed and stable. In fact the tooling may be subject to important changes, like the necessity to replace a model editor by another or the necessity to take into account new features. This study concentrates on the agility for developing and then maintaining VoiceBench which is the framework developed for developing interactive voice services (Section 3.3.2)..

The hypothesis we want to verify is:

H3: *Service creation tools exploiting MDA are easy to maintain*

We developed different variants of the VoiceBench tooling depending on the UML tool used for editing models. The initial implementation was based on Telelogic TAU Tool,

then a second implementation was based on Objecteering from Softeam which reused most of all components of the primary implementation. Then we reaslized third variant implementation using RSM from IBM. For the latest variant, a specific simulator executing in RSM was developed as a remplacement of the IF simulator in the initial implementation.

As part of the iterative development of the tool chain, for the purpose of debugging and demonstration, we developed also a list of "Toy services": TV Recorder, Coffee, and AlloCine.

4.2.3.2 Realization

Figure 20 (in Section 3.3.2), depicts overall architecture of the VoiceBench framework.

Figure 32 below summarizes the development activities and estimated costs for developing the tool chain.

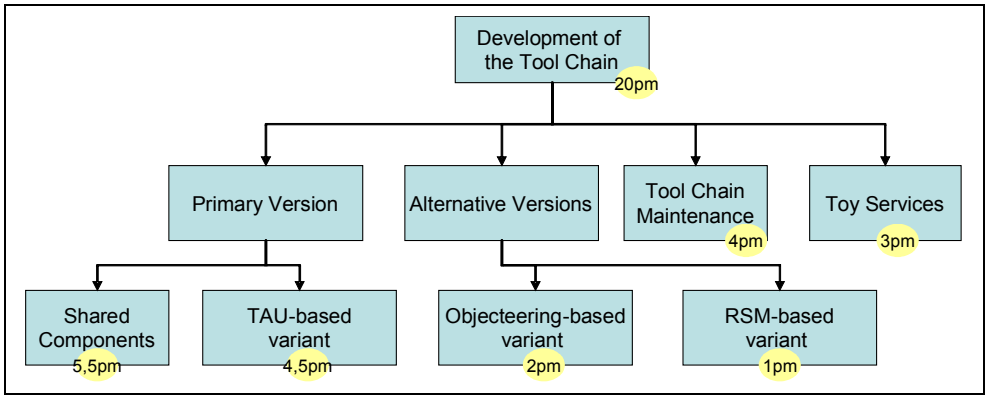


Figure 32: Split of activities for tool chain development

The shared components are the software or design components that are shared by all the variants: these are the metamodel definition, the textual syntax definition, the text to metamodel parser, the voice metamodel to code generator, the IF-based voice simulator tool.

Each variant of the MDD tool chain is characterized by a different UML base tool and, consequently by a different implementation of the UML profile and associated transformers.

4.2.3.3 Evaluation and lessons learned

4.2.3.3.1 Hypothesis and threads to validity

The hypothesis to be verified is:

H3: *Service creation tools built with MDA are difficult to develop but easy to maintain*.

The details of measures of quantitative evaluation are provided in Annex B.

When evaluating the efforts for developing and maintaining the MDD tool chain, the following costs were considered:

- The cost for developing the MDD Tool chain
- The cost for replacing the modelling tool in the MDD tool chain (like using RSA from IBM in place of Telelogic TAU)
- The cost for the maintenance of the MDD tool chain (like changing the metamodel).

The following *threads to validity* need to be considered:

- The development of the tool chain was done by MDA experts. Hence the overhead cost for learning how to apply effectively MDA technologies (like design of metamodels and design of transformations) may be under-estimated.
- The technology for creating DSLs with model-oriented techniques is evolving towards solutions that tend to automatize as much as possible the connection between abstract syntaxes and concrete syntaxes [Muller04]. In our experiment support for textual notation was realized without assistance of such kind of tools (e.g the grammar was defined and implemented with traditional lex/yacc tooling).

4.2.3.3.2 Effort for developing the MDD tool chain (initial version):

Within the list of components involved in the MDD tool chain some of them are external pre-existing tools and hence cannot count in the cost of the development of the tool. The components that have an effective contribution in the cost are the specific UML profiles and all the transformers that allow making the integration between the various tools.

In the overall cost we need to include the definition of the abstractions that are used by the developed software: in this category the most important is the cost for developing the *Voice metamodel* on which all the software is based.

The metamodel plays a central role because it is used to generate the concrete XML schema for storing the telecom service definitions. In the case of our telecom domain, the metamodel represents an executable language with all the needed computational details, such as the capability to express arbitrary actions and expressions. The metamodel was defined iteratively, in parallel with the implementation of the transformations. One of the lessons learned from this project is that it is not realistic to design a metamodel of an executable language without implementing in parallel a list of tools that make use of it. Implementing the metamodel means, in our context, that we are capable of storing complete telecom specifications using the XML schema automatically generated from the metamodel, and also to implement the transformations that allow executing the telecom

models in a target execution platform. The most common problems found during metamodel development were:

- Incompleteness of the metamodel: as one goes along in the implementation, we discover that some aspects of the intended functionality cannot be captured: new attributes or new classes are added.
- Difficulty to structure the metamodel in the way that provides good compromise between reuse and readability.

Three major versions of the metamodel were produced during the project before obtaining a stabilized version.

Another important design artefact is the definition of the textual syntax that corresponds to the metamodel. The textual notation with the corresponding textual to metamodel parser simplifies the task of connecting UML tools to the Telecom engine, since it is in general easier to produce a compact textual notation than to produce a metamodel XMI rendering. This textual syntax was used in the Objecteering version but could be used for other future implementations.

The table below provides the cost of each all conceptual or software components that are needed in the setup of the MDD Tool. We have distinguished the costs that were common to all the variants of the MDD tool chain from those that were specific to the version under consideration in this section.

Common costs:

The definition of the VOICE Metamodel	1,5 person/month
The definition of the VOICE textual syntax	0,2 person/month
Text to metamodel translator [TELECOMTXT2MM]	0,5 person/month
Production of the executable VOICE logic [VOICE2CODE]	0,8 person/month
The Simulator based on IF [IF_SIM]	2,5 person/month
TOTAL	5,5 person/month

Table 1 : Costs for shared components

Costs specific to the TAU version

The Voice profile on top of Telelogic TAU [PROF_TAU]	2 person/month
Translation Voice to IF [TAU2IF]	1,5 person/months
Export of UML/TAU in terms of Voice metamodel [TAU2VOICE]	1 person/month

TOTAL	4,5 person/month

Table 2 : Costs for TAU-based version

The following table gives the total cost for the developing of the MDD tool chain in the TAU configuration.

Shared components	5,5 person/month
Specific components of the TAU Version	4,5 person/month
TOTAL	10 person/month

Table 3 : Summary of costs for tool chain development

These measures shows that setup of a simulation facility and development of the metamodel take a significative part in the consumption of resources.

4.2.3.3.3 The cost for replacing the modelling tool in the MDD tool chain

Two variants of the initial tool chain were build, one on top of Objecteering and another on top of IBM/RSM. We are providing here the numbers for the former.

Costs specific to the Objecteering version

The Voice profile on top of Objecteering [PROF_OBJ]	1,2 person/month
Export of UML/Objecteering in terms of Voice metamodel [OBJ2VOICE]	0,9 person/month
TOTAL	2 person/month

Table 4 : Costs for Objecteering-based based version

The difference between the cost of development of the voice profile on top of Telelogic and the corresponding version on top of Objecteering can be explained by two major reasons: To satisfy request from users, we developed on top of the TAU version a rich dedicated GUI to define voice messages, whereas in the Objecteering version we simply used the standard editor for UML notes. Another reason for the difference in the cost is that the implementation language for the TAU version is C++, whereas the Objecteering version used a dedicated model manipulation language named J, which simplified a lot the development of the profile and corresponding behaviour rules.

We have here typically the kind of compromise we have to do when customizing an existing modelling tool: in particular we need to pay attention on (1) the quality of the GUI that the CASE tool offers "for free", (2) the cost of enriching the GUI interface. The actual users of the Voice MDD tool chain were in fact very sensitive to the GUI aspects, mostly because they were not people with programming profile.

The cost for replacing the modelling tool was then very low in this case (only 2 man/month). However it needs to be moderated by the *perceived quality of the tool chain* (dependent variable) from end-users which did not found all the functionality that was in the primary version. We have estimated the cost needed to develop the equivalent interface for voice messages in the new modelling tool to 0,5 person month and estimated the cost to provide the equivalent syntax and semantic check in expressions to 1,5 person month. The table below provides the cost for replacing the modelling tool with the minimal support and with the complete support (adding estimated efforts for the missing functionalities).

Cost for replacing the modelling tool (minimal support)	2,0 person/month
Cost for replacing the modelling tool (complete support)	4,0 person/month

Table 5 : Summary of costs for modelling tool substitution

One interesting conclusion of this study is that it may be very cheap to substitute one UML tool by another if the level of the exigency is low in terms of graphical interface, something that can be acceptable to certain kind of skill users that do need too much assistance.

4.2.3.3.4 The cost of maintaining the MDD tool chain

There can be various reasons for having maintenance activities on the MDD tool chain. Among them we have:

(i) Discovering of a bug when trying to use a functionality which was not sufficiently tested in all situations.

(ii) A release update in the tools that are being used, like

- A new version for the supporting VoiceXML gateway, which requires a change in the execution engine

- A new version of the UML tool being used which requires upgrading the implemented profile

(iii) A request for a minor functional improvement (we do not consider major enhancement as a maintenance activity).

The table below gives the occurrence of these activities and their global contribution in the maintenance cost observed in one year, after the first version of the MDD tool chain was produced.

	Occurrences	**Cost (person/months)**	**Maintenance Cost (%)**

Bug fixing	22	2,4	60%
Upgrades of supporting tools	2	0,4	10%
Improvement requests	5	1,2	30%
TOTAL		4,0	

Table 6 : Costs for tool chain maintenance

Bug fixing was the most important maintenance activity because the tool chain was still young. In general each new service development brings their new list of discovered bugs as well as a new list of improvement requests.

As a partial conclusion, we can say that the maintenance cost for a MDD tool chain is relatively high. The complexity of the MDD tool chain, which involves the integration of various independent tools, is indeed one of the major reasons for this.

4.2.3.3.5 Hypothesis verification

Our conclusion is that the hypothesis is not really verified. It would be probably more appropriate to say: *Service creation tools built with MDA are relatively easy to develop but hard to maintain*.

They are especially easy to develop if we limit to basic functionality like graphical editing, textual editing and code generation. Support of simulation is generally more complex. Evolution of metamodels and proper handling of their impact remains a problem in terms of maintenance effort.

4.3 Validation Summary

In this section we summarize our conclusions regarding the three hypotheses:

> H1: *Use of MDA in service creation tools enhances productivity of service designers and service developers*
> H2: *Explicit modelling of service logic facilitates service evolution, even for complex services*.
> H3: Service creation tools exploiting MDA are difficult to develop but easy to maintain.

Our experiments provides elements towards confirming that model-driven tool chain allows making significant productivity gain to service designers and service developers (hypothesis H1). The measures presented in Section 4.2.1 concerning the development of a voice service development showed that it was possible to obtain 25 % of productivity gain in design activities and 70 % of productivity gain in implementation activities thanks to the usage of the MDD tool chain.

Also, as illustrated by Dinner Planning experiment (Section 4.2.2) domain specific language used to model service behaviour (like SPATEL) combined with a model-aware service creation environment (like SPATEL Engine) can favour agility, especially to ensure evolution of composite services. We consider hypothesis H2 to be verified.

However the construction and maintenance of MDD tool chains is not made for free. According to our measures described in Section 4.2.3, development and maintenance of the VoiceBench framework cost approximately ~1 person year in terms of resources and still required approximately 0.4 person year for maintenance. Hypothesis H3 is not verified.

5 Chapter - Conclusion and Perspectives

5.1 Context of work: MDA and platform modernization

The telecommunication industry tends to use more and more the technologies that come from the IT industry. This evolution has indeed been accelerated with the growth of internet. Voice-based telecommunication services offer a good illustration of this tendency: the VoiceXML W3C standard has made possible to develop an interactive voice application in a similar way as a web application is developed.

A similar situation comes with telecom composite services integrating communication facilities (like messaging, presence, and so on) with internet facilities (translation, weather, news and so on). The adoption by the industry of SOA technologies and the related standards (like SOAP or REST web services) represents a major step to enable effective sharing and integration of software resources. The tendency is to create services "in the cloud" that clients can access by means of programmatic web interfaces invoked from desktop applications of mini-applications installed in smartphones (Iphone, Android phones and so on).

Thanks to web service technology, telecom operators can offer to third party developers simplified access to communication facilities to help them create added-value services exploiting their network capacities.

Usage of modern IT middleware platforms to implement telecom services is an essential step to gain better control over development and maintenance costs. But, unfortunately, this is not always sufficient. This is where MDD intervenes. The role of MDD is to fill the gap between design and execution, and more specifically, between domain-specific design languages (the VOICE and SPATEL) and the execution platforms. Modelling and code generation are key technologies to realize the bridge between design and execution.

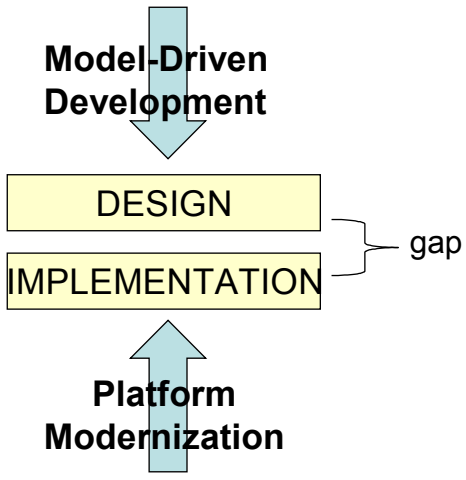


Figure 33: Lowering the gap between design and implemtation

To summarize, there are two complementary pressures that potentially can contribute significantly to increase agility for building and maintaining telecommunication services. One is "platform modernization" – which is illustrated by the advent of VoiceXML and SOA Web Services. The other is "model driven development (MDD)" – which, in our case, is illustrated by the definition of DSLs using meta-modelling, and by the creation of execution frameworks that operate on "models" (SPATEL Engine and VoiceBench), through the development of transformations that automatizes large amounts of the effort needed to deploy and test services.

5.2 Summary of defended thesis and contribution

In the state of the art section we pointed out various interesting research efforts that attempt to combine the benefits of service oriented architecture (like easy integration of external software) with those bring by model-driven development (like separation of concerns and productivity gain). These studies, however, rarely try to consider all aspects of service specification, that is to say, not only interface and behaviour definition, possibly complemented with semantics and QoS characteristics, but also user interaction (through GUI or voice).

Moreover, based on experiments made internally at France Telecom we know that applying model driven technology to service development *may be risky* due to the inherent complexity of MDA - like dealing with various levels of abstraction - and because of the potential overhead it generates in the development process - such as the necessity to develop specific transformations to adapt model-based representations into legacy XML formalisms supported in middleware platforms used by telecom companies.

Nevertheless, our defended thesis is that combination of MDA with SOA can *significantly* improve agility of telecom service creation if MDA is appropriately applied. Firstly, we recommend implementing a "MDA flavoured" host-target approach, similar to host-target testing process used for developing embedded systems, which in our case will consist of three main principles:

- Using a model centric high-level service specification formalism (a DSL) targeting the special requirements of telecommunication services (long-running, event-based, server/terminal distinction, QoS).

- Exploiting a native execution framework (the host and default target environment) that allows incremental and iterative development of a service thanks to early simulation and testing.

- Automating as much as possible the production of implementations in alternative target environments (at server and terminal side) thanks to the development of various model transformers generating code and test procedures.

The second essential point for an *effective* agility is that MDA appliance need to be pragmatic as opposed to dogmatic: two examples of crucial decisions to be taken are

selection between metamodel or uml profile technology to define DSLs or the selection between direct code generation and model to model transformation (see Section 3.4.3).

Last but not least, for an effective agility, the developed concrete artefacts (formalisms, methods and tools) to support our vision need to efficiently match domain requirements and specificities. This concerns the three identified aspects of our MDA flavoured "host-target" approach (high-level DSLs, native framework and transformations).

Regarding DSLs, for voice based telecommunication services we defined the VOICE metamodel and accompanying notation to model typical stateful voice dialog logic (section 3.3.1). For integrated telecommunication services, we proposed a methodology for building composite services (section 3.1.3) and proposed the SPATEL formalism to cope with arbitrarily complex services. One distinctive characteristic of this formalism is the ability to integrate different aspects of service specification (behaviour, semantics, non functional behaviour and multi-modal user interaction).

Regarding the native execution framework, one important contribution in our work is the support of implementation variability (section 3.2.3.2) to allow quick replacement of service components. This is especially useful for incremental simulation.

Finally regarding the last kind of artefacts - transformer components, we should point out that their design was one of the leading motivations for us to contribute in the development of a model to model transformation standard having *imperative characteristics* (see QVT Operational in section 2.2.2.3). Indeed most of the transformations we developed were sufficiently complex to eliminate the possibility for us to use pure declarative transformation techniques.

To validate our work we have conducted various experiments (section 4.2) that provided elements towards confirming that *use of MDA in service creation tools enhances productivity of service designers and service developers,* that *explicit modelling of service logic facilitates service evolution,* but in the other hand highlighted the fact that a model-driven tool chain remain difficult to maintain. These conclusions emphasize our belief in the importance of pragmatic MDA appliance to mitigate potential risks and to take the best profit of it.

5.3 Perspectives

In terms of perspectives related to our work, we would like to mention three ongoing activities:

- Enhancing SPATEL with natural language annotations to facilitate service composition based on the interpretation of natural language
- TelcoML standardization effort at the OMG
- Full support of multi-modality

5.3.1 TelcoML standardization effort

We have emphasized in this document the importance of standardization in service development: it mitigates dependencies on tool vendors and increases reuse opportunities of service descriptions. We are currently involved at the OMG in the process of responding to a Request for Proposal (RFP) for standardizing a modelling language for telecommunication service development (see [omg-telco]).

Our proposal called TelcoML, which is being submitted with other Telco and IT partners like AT&T, IBM and HP, will be a specialization of SoaML [omg-soaml] with specific concepts taken from SPATEL, in particular those dedicated to voice interaction modelling, semantic annotations and extensions for service composition. Besides that a library of telecom enablers will be defined in the form of SoaML service interfaces to facilitate interoperability of added-value composite services that exploit communication facilities provided by telecom operators. This library will include also management operations in line with NGOSS architecture [tmf-ngoss] defined by TeleManagement Forum.

5.3.2 Full support for Multi-Modality

The SPATEL language allows to describe services in which voice interaction can be mixed with GUI based interaction, thanks to explicit support of these two interaction modes (see Sections 3.2.2.3 and 3.2.2.4). However the impact of integrating various kinds of multi-modality appliance (redundancy, complement, sequence and synergy, according to [Nigay94]) in service design was not examined in detail and is left for further study.

5.3.3 Model based Natural Language annotations

In order to allow end-users to create simple and personalized composite services we have created a tool that interprets user requests in natural language and produces as output an orchestration script (in SPATEL) chaining the service invocations that fulfils the request. The details on this ongoing experiment are provided in Annex C.

In the actual prototype, *natural language annotations* (that's to say, the vocabulary and the syntax patterns to match a service included in the catalogue) are defined in an ad-hoc manner through configuration files and direct coding of rules.

However if we are able to model properly natural language characteristics and express them as annotations of SPATEL model elements - at same level than semantic annotations - then we could consider application of MDA techniques to generate the code of the rules. The integration of new service components in the interpretation system, which still remains a challenge, would become faster thanks to model based formalization.

6 Bibliography/References

Scientific Papers

[Aalst03] W. van der Aalst, “Don’t go with the flow: Web services composition standards exposed,” IEEE Intelligent Systems, vol. 18, pp. 72–76, 2003.

[Andrews00] G. Andrews - "Foundations of Multithreaded, Parallel, and Distributed Programming", 2000, Addison–Wesley, ISBN 0-201-35752-6. Chapter 7 & Chapter 8.

[AspectJ02] AspectJ Team - "The AspectJ programming guide". http://www.eclipse.org/aspectj/doc/released/progguide,2002-2003.

[Baravaglio05] Alberto Baravaglio , Carlo Alberto Licciardi , Claudio Venezia, Web Service Applicability in Telecommunication Service Platforms, Proceedings of the International Conference on Next Generation Web Services Practices, p.39, August 22-26, 2005

[Bauer04] Bernhard Bauer and Jörg P. Müller - MDA Applied: From Sequence Diagrams to Web Service Choreography Lecture Notes in Computer Science, 2004, Volume 3140/2004, 779, DOI: 10.1007/978-3-540-27834-4_16

[Beck02] Kent Beck - "eXtreme Programming - The reference", 2002 ISBN 2-7440-1433-8

[Belaunde02] M. Belaunde, J.P Almeida, J. Pires, M. Born et al, Modatel Project - "Assessment of the Model Driven Technologies –Foundations and Key Technologies". Section 2.1.4.1. http://www.modatel.org/~Modatel/pub/deliverables/D2.1-final.pdf

[Belaunde99] A Pragmatic Approach for Building a Flexible UML Model Repository. UML 1999 conference: pages 188-203.

[Belouadha10] Fatima-Zahra Belouadha, Hajar Omrana and Ounsa Roudiès - A model-driven approach for composing SAWSDL semantic Web services, IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 2, No 1, March 2010 ISSN: 1694-0814

[Berners01] Berners-Lee, Tim; James Hendler and Ora Lassila (May 17, 2001). "The Semantic Web". Scientific American Magazine. http://www.sciam.com/article.cfm?id=the-semantic-web&print=true. Retrieved March 26, 2008.

[Bezivin03] J Bézivin, G Dupé, F Jouault, G Pitette, Jamal Eddine, Rougui - "First experiments with the ATL model transformation language: Transforming XSLT into XQuery". 2nd OOPSLA Workshop, 2003.

[Blow04] M.Blow,Y.Goland,M.Kloppmann,F.Leymann,G.Pfau,D.Roller,M.Rowley - BPELJ - BPEL for Java - A Joint White Paper by BEA and IBM March 2004 Available at: http://download.boulder.ibm.com/ibmdl/pub/software/dw/webservices/ws-bpelj/ws-bpelj.pdf

[Bozga04] M.Bozga, S.Graf, I.Ober, Iulian Ober and J. Sifakis - Tools and Applications II: The IF Toolset - Proceedings of SFM'04 (Bertinoro, Italy), September, 2004 LNCS vol. 3185, Springer-Verlag

[Brockmans06] S Brockmans, P Haase, P Hitzler, R Studer - "A Metamodel and UML Profile for Rule-Extended OWL DL Ontologies" The Semantic Web: Research and Applications. Lecture Notes in Computer Science, 2006, Volume 4011/2006, 303-316, DOI: 10.1007/11762256_24

[Cano] Model-driven development of embedded systems on OSGi platform, Julio Cano, Natividad Martínez Madrid, Ralf Seepold, Universidad Carlos III de Madrid http://www.martes-itea.org/public/papers/cano.pdf

[Chabeb08] Y.Chabeb and S.Tata - Yet Another Semantic Annotation IADIS International Conference WWW/Internet 2008 Available at: http://picoforge.int-evry.fr/projects/svn/soc/YET-ANOTHER-SEMANTIC-ANNOTATION-FOR-WSDL.pdf

[Clarke05] Siobhan Clarke and Elisa Baniassad - "Aspect-oriented analysis and design: The Theme approach". Addison-Wesley 2005.

[Combemale08] Benoit Combemale. Approche de métamodélisation pour la simulation et la vérification de modèle. Thèse de doctorat, Institut National Polytechnique de Toulouse, juillet 2008. Available at: http://ethesis.inp-toulouse.fr/archive/00000666/

[Desfray00] Ph. Desfray - "UML Profiles versus Metamodel extensions : An ongoing debate". http://www.omg.org/news/meetings/workshops/presentations/uml_presentations/5-3%20Desfray%20-%20UMLWorkshop.pdf

[Duddy03] Keith Duddy, Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel - Model Transformation: A declarative, reusable patterns approach 10.1109/EDOC.2003.1233847, September 2003, ISBN: 0-7695-1994-6

[Dumez08] Christophe Dumez, Jaafar Gaber and Maxime Wack - Web services composition using UML-S: a case study 4th international conference on Next generation Web Services Practices (NWeSP'08)

[Emmen02] , Designing knowledge management systems using XML, XSLT, and MPEG-7, Ad Emmen. Pages: 485 - 487, 2002, ISSN:0928-7329

[Fenster10] L. Fenster, B. Hamilton UML or DSL: Which Bear Is Best?, March 2010. http://msdn.microsoft.com/en-us/architecture/ff476944.aspx

[Fielding00] Roy Fielding - "Representational State Transfer (REST), Chapter 5". Available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[Fowler01] M. Fowler and J.Highsmith - "The Agile Manifesto", August 2001. http://andrey.hristov.com/fht-stuttgart/The_Agile_Manifesto_SDMagazine.pdf

[France04] R. France, I. Ray, G. Georg, S. Ghosh - "Aspect-oriented approach to early design modelling". Software, IEE Proceedings -, 151(4):173-185, 8 2004.

[Gasevic06] Model Driven Architecture and Ontology Development, Dragan Gašević http://www.springer.com/3-540-32180-2

[Gronmo04] Grønmo, R., Solheim, I.: Towards Modelling Web Service Composition in UML. In Proceedings of WSMAI-2004, INSTICC Press 2004, pp. 72-86.

[Guerbi09] Tahar Gherbi , Djamel Meslati , Isabelle Borne - MDE between Promises and Challenges, UKSim 2009: 11th International Conference on Computer Modelling and Simulation . March 2009, pp. 152-155

[Guizzardi02] G. Guizzardi, L. Ferreira Pires and M. van Sinderen. On the role of Domain Ontologies in the Design of Domain-Specific Visual Languages, 2nd Workshop on Domain-Specific Visual Languages. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002), Seattle, Washington, USA, 2002.
Available at http://wwwhome.cs.utwente.nl/~guizzard/oopsla-dsvl.pdf

[Harel00] D. Harel and B. Rumpe. Modelling Languages: Syntax, Semantics and All That Stuff, Technical Report, The Weizmann Institute of Science, Rehovot, Israel, MCS00-16, 2000.

[Harel87] David Harel, Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8(3):231–274, June 1987.
Available at:
http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf

[Harel96] D. Harel, E. Gery - "Executable object modeling with statecharts". International Conference on Software Engineering archive.
Proceedings of the 18th international conference on Software engineering table of contents, Berlin, Germany, Pages: 246 - 257, ISBN:0-8186-7246-3

[Holzmann91] Holzmann, G.J., “Design and Validation of Communication Protocols”, Prentice Hall, 1991.

[Ipl96] IPL Information Processing Ltd., "Host Target testing". Available at: http://www.ipl.com/pdf/p0822.pdf

[Jezequel08] .J.M Jezequel - "Model driven design and aspect weaving". Journal of Software and Systems Modeling (SoSyM), 7(2):209--218, May 2008

[Kobryn00] C. Kobryn - Architectural patterns for metamodeling: the Hitchhiker's guide to the UML metaverse. October 2000
UML'00: Proceedings of the 3rd international conference on The unified modelling language: advancing the standard. Pages: 497-497. ISBN ~ ISSN:0302-9743 , 3-540-41133-X. Springer-Verlag.

[Korp02] O.Kopp, F.Leymann - Choreography Design Using WS-BPEL
Available at: http://sites.computer.org/debull/A08Sept/kopp.pdf

[Kruchten99] Philippe Kruchten, *Rational Unified Process-An Introduction,* Addison-Wesley, 1999

[Kurtev02] Kurtev, I., Bézivin, J., Aksit, M.: Technical spaces: An initial appraisal. In: CoopIS, DOA’2002 Federated Conferences, Industrial Track. (2002)
Available at:
https://gforge.inria.fr/scm/viewvc.php/*checkout*/Publications/Before2009/PositionPaperKurtev.pdf?root=atlantic-zoos

[Lin09] Sheng-Shi Lin, Shin-Shing Shin, Ming-Che Hsieh, Jen-Her Wu, Wei-Sheng Hung, "MDA-Based UI Modeling and Transformation of Spoken Dialog Systems," his, vol. 1, pp.47-51, 2009 Ninth International Conference on Hybrid Intelligent Systems, 2009

[Mahmoud05] Qusay H. Mahmoud - "Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)".
http://www.oracle.com/technetwork/articles/javase/index-142519.html

[Martens05] W. Martens, F. Neven, T. Schwentick, G. Beck - "Expressiveness and Complexity of XML Schema". ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY, Pages 1-42.

[Martin91] J.Martin - "Rapid Application Development", 1991
ISBN 0-02-376775-8

[McNeile03] A.McNeile - "MDA: The Vision with the Hole?", 2003.
Available at: http://www.metamaxim.com/download/documents/MDAv1.pdf

[Mellor02] S.Mellor, M. Balcer - "Executable UML: A foundation for model-driven architecture", chapter 1.2 Executable UML, Addison Wesley, 2002

[Mens05] Tom Mens, Krzysztof Czarnecki and Pieter Van Gorp - "A Taxonomy of Model Transformations", Dagstuhl Seminar on Language Engineering for Model-Driven Software Development, 2005.
Available at: http://drops.dagstuhl.de/opus/volltexte/2005/11/

[Micskei10] Z. Micskei and H. Waeselynck: The many meanings of UML 2 Sequence Diagrams: a survey, Software and Systems Modeling, Springer, Online first, DOI:10.1007/s10270-010-0157-9, 2010,
http://springerlink.metapress.com/content/6716hk1844h16694/

[Milanovic09] Milan Milanovic1,Dragan Gasevic2, Adrian Giurca3,,Gerd Wagner3 and Vladan Devedzic1 - Bridging concrete and abstract syntaxes in model-driven engineering: a case of rule languages. Softw. Pract. Exper. 2009; **39**:1313–1346

[Moreno07] N. Moreno, P. Fraternali, A. Vallecillo. "WebML Modelling in UML". IET Software 1(3):67-80, 2007.

[Muller04] P.A Muller, Ph. Studer, J.M Jezequel - Model-driven generative approach for concrete syntax composition. Proceedings of Oopsala 2004 conference.
Available at: http://www.softmetaware.com/oopsla2004/muller.pdf

[Muller05] P.A. Muller, F. Fleurey, J.M. Jezequel - "Weaving Executability into Object-Oriented Meta-languages".
Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, 2005, Volume 3713/2005, 264-278, DOI: 10.1007/11557432_19.

[MullerFleurey05] P.A Muller, Franck Fleurey, Didier Vojtisek, Z. Drey, D. Pollet, F.Fondement, P.Studer, J.M. Jézéquel - On Executable Meta-Languages applied to Model Transformations.
http://www.irisa.fr/triskell/publis/2005/Muller05c.pdf

[Nigay96] Laurence Nigay, Joelle Coutaz. - Les propriétés "CARE" dans les Interfaces multimodales. Actes de la conférence IHM'94, Lille, 1994

[Patrascoiu04] Octavian Patrascoiu, YATL:Yet Another Transformation Language. In Proceedings of the 1st European MDA Workshop, MDA-IA, pages 83-90. University of Twente, the Nederlands, January 2004.
Available at: http://www.cs.kent.ac.uk/pubs/2004/1829/content.pdf

[Quartel04] D. Quartel, R. Dijkman, M. van Sinderen - Methodological Support for Service-oriented Design with ISDL. Proceedings of the 2nd international conference on Service oriented computing. New York, NY, USA. Pages: 1 - 10 Year of Publication: 2004 ISBN:1-58113-871-7

[Quartel07] D. Quartel, M. Steen & S. Pokraev, M. van Sinderen - COSMO: A conceptual framework for service modelling and refinement.
Inf Syst Front (2007) 9:225–244 DOI 10.1007/s10796-007-9034-7

[Sendall03] Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software (2003) 42–45
Available at: http://lgl.epfl.ch/pub/Papers/sendall-tech-report-EPFL-model-trans.pdf

[Shani08] U. Shani, A. Sela - Software design using UML for empowering end-users with an external domain specific language. International Conference on Software Engineering. Proceedings of the 4th international workshop on End-user software engineering. Pages: 52-55, 2008. ISBN:978-1-60558-034-0
http://doi.acm.org/10.1145/1370847.1370859

[Simonin10] J. Simonin - Thèse: Conception de l'architecture d'un système dirigée par un modèle d'urbanisme fonctionnel.
Available at: http://hal.inria.fr/tel-00512182

[Sinderen02] M. van Sinderen, L. Ferreira Pires, G. Guizzardi. - Design of Domain-Specific Visual Languages,
2nd Workshop.17th ACM Conference on Object-Oriented Programming, Systems, (OOPSLA 2002), Seattle, Washington, USA, 2002.
Available at http://wwwhome.cs.utwente.nl/~guizzard/oopsla-dsvl.pdf

[Strassner04] J. Strassner, J. Fleck, J. Huang, C. Faurier, T. Richarson - TMF White Paper on NGOSS and MDA, 2004.
Available at: http://www.bptrends.com/publicationfiles/04-04%20WP%20TMF%20MDA-NOGSS%20-%20Strassner%20et%20al.pdf

[Sunye01] G. Sunyé, F. Pennaneac'h, W.M Ho, A. Le Guennec, J.M Jézéquel - "Using UML Action Semantics for Executable Modeling and Beyond".
Advanced Information Systems Engineering.Lecture Notes in Computer Science, 2001, Volume 2068/2001, 433-447, DOI: 10.1007/3-540-45341-5_29

[Vallecillo04] L. Fuentes, A. Vallecillo - "An Introduction to UML Profiles".
UPGRADE, The European Journal for the Informatics Professional, 5(2):5-13, April 2004. ISSN: 1684-5285.
http://www.lcc.uma.es/~av/Publicaciones/04/UMLProfiles-Upgrade04.pdf

[Varro02] D. Varro, G. Varro and A. Pataricza. Designing the automatic transformation of visual languages. Science of Computer Programming, vol. 44(2):pp. 205--227, 2002.

[Vigneras08] Pierre Vigneras - "Why BPEL is not the holy grail for BPM". Oct 21, 2008
Available at: http://www.infoq.com/articles/bpelbpm

[Villalonga07] C.Villalonga, M.Strohbach, N.Snoeck, M.Sutterer, M.Belaunde, E.Kovacs, A.Zhdanova, L.W. Goix, O.Droegehorn: "*Mobile Ontology: Towards a Standardized Semantic Model for the Mobile Domain*". ICSOC Workshops 2007: 248-257

[Volkmann02] M.Volkmann, Axis - an open source web service toolkit for Java. http://www.ociweb.com/javasig/knowledgebase/2002Sep/

[Willink03] E. D. Willink. UMLX: A graphical transformation language for MDA. Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, University of Twente, Enschede, The Netherlands, June 26-27, 2003, CTIT Technical Report TR–CTIT–03–27, University of Twente, 2003, http://trese.cs.utwente.nl/mdafa2003pp. 13-24.

[Zhu09] Zhengdong Zhu Ronggui Lan Ruifang Ma Yanping Chen - E-Business and Information System Security, 2009. EBISS '09. International Conference on, 23-24 May 2009, 978-1-4244-2909-7 http://www.computer.org/portal/web/csdl/doi/10.1109/SCC.2010.28

Industry Standards

[itu-odp] ISO/IEC IS 10746 | ITU-T X.900 - "Open Distributed Processing Reference Model"

[itu-sdl] ITU-T Recommendation Z.100, Annex F: *SDL Formal Semantics Definition*, International Telecommunications Union (ITU), Geneva, 2000.

[jsr-000315] JSR-000315 Java™ Servlet 3.0 Specification. http://jcp.org/aboutJava/communityprocess/final/jsr315/

[oasis-bpel] Organization for the Advancement of Structured Information Standards (OASIS) - Web Services Business Process Execution Language (WSBPEL) http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[oasis-pbel4p] Organization for the Advancement of Structured Information Standards (OASIS) - BPEL for People http://www.oasis-open.org/committees/bpel4people/charter.php

[oasis-rm] Organization for the Advancement of Structured Information Standards (OASIS) - "Reference Model for Service Oriented Architecture" 1.0, Oct 12, 2006, page 8. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[omg-bpmn] Object Management Group - "Business Process Modeling Notation" http://www.omg.org/spec/BPMN/1.2/

[omg-corba] Object Management group - "Common Object Request Broker Architecture" http://www.omg.org/spec/CORBA/2.0/

[omg-m2t] Object Management Group - "MOF Models to Text Transformation Language v1.0" Release date: January 2008 http://www.omg.org/spec/MOFM2T/1.0

[omg-mda] Object Management Group - Object Management Group, Model Driven Architecture (MDA), ormsc/01-07-01, July 2001

[omg-mdag] Object Management Group - "MDA Guide, version 1.0.1". http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf

[omg-mof] Object Management Group - "Meta Object Facility Core, v2.0"
Release date: January 2006
http://www.omg.org/spec/MOF

[omg-qvt] Object Management Group - "Meta Object Facility 2.0 Query/View/Transformation v1.1 - Beta 2"
Release date: Dec 2009
http://www.omg.org/spec/QVT/1.1/Beta2

[omg-telco] Object Management Group - UML Profile for Advanced and Integrated Telecommunication Services RFP
http://www.omg.org/techprocess/meetings/schedule/UML_Profile_for_Advanced_and_Integrated_Telecommunication_Services_RFP.html.

[omg-uml] Object Management Group - "Unified Modeling Language, v2.3"
Release date: May 2010
http://www.omg.org/spec/UML/2.3

[omg-xmi] Object Management Group - "MOF 2.0/XMI Mapping, v2.1.1"
Release date: December 2007
http://www.omg.org/spec/XMI/2.1.1

[parlay] Parlay X - Web service APIs for the telephone network
http://docbox.etsi.org/TISPAN/Open/OSA/ParlayX30.html

[tmf-etom] TMF, “GB921: eTOM – the Business Process Framework, version 3.5”, July 2003.

[tmf-sid] Tele Management Forum, "GB922, Information Framework (SID) Suite", Release 9.0 , 2010
http://www.tmforum.org/InformationFramework/

[w3c-ccxml] World Wide Corsortium - Voice Browser Call Control: CCXML Version 1.0
W3C Candidate Recommendation 1 April 2010
http://www.w3.org/TR/ccxml/

[w3c-owl] World Wide Corsortium - OWL 2 Web Ontology Language Document Overview.
W3C Recommendation 27 October 2009
http://www.w3.org/TR/owl2-overview/

[w3c-rdf] World Wide Corsortium - Resource Description Framework (RDF).
Publication date: 2004-02-10
http://www.w3.org/RDF/

[w3c-sawsdl] World Wide Corsortium - Semantic Annotations for WSDL and XML Schema.
W3C Recommendation 28 August 2007
http://www.w3.org/TR/sawsdl/

[w3c-soap] World Wide Corsortium - SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) W3C Recommendation 27 April 2007
http://www.w3.org/TR/soap12-part1/

[w3c-vxml] World Wide Web consortium - "Voice Extensible Markup Language (VoiceXML) Version 2.0"
W3C Recommendation 16 March 2004
http://www.w3.org/TR/voicexml20/

[w3c-wsdl] World Wide Web consortium - "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language"
W3C Recommendation 26 June 2007
http://www.w3.org/TR/wsdl20/

[w3c-xpath] World Wide Corsortium - XML Path Language (XPath) Version 1.0
W3C Recommendation 16 November 1999
http://www.w3.org/TR/xpath/

[w3c-xslt] World Wide Web consortium - "XSL Transformations (XSLT) Version 1.0"
W3C Recommendation 16 November 1999
http://www.w3.org/TR/xslt

Tools

[asterisk] The Open Source Telephony project
http://www.asterisk.org/

[cscxml] SCXML, “The jakarta project commons SCXML"
http://jakarta.apache.org/commons/scxml/, 2006.

[django] The Django framework - http://www.djangoproject.com/

[emf] Eclipse Modelling Framework. http://www.eclipse.org/emf

[gcalendar] Google Calendar.
http://www.google.com/intl/fr/googlecalendar/event_publisher_guide.html

[gcc] GNU C Compiler.
http://gcc.gnu.org

[jython] Jython: Python for the Java Platform.
http://www.jython.org/

[metaedit] MetaEdit+ Domain-Specific Modeling environment
http://www.metacase.com/MetaEdit.html

[objecteering] The Model Driven Development Tool
http://www.objecteering.com/

[python] Python Programming Language - http://python.org

[rsm] Rational Software Architect
http://www.ibm.com/developerworks/rational/products/rsa/

[smartqvt] SmartQVT : Open Source Implementation of QVT Operational language.
http://sourceforge.net/projects/smartqvt/

[symbian-s60] Symbian S60 - Operating System for Smartphones -
http://www.symbian.org/

[tau] Telelogic TAU UML Suite.
http://www-01.ibm.com/software/awdtools/tau/

[tomcat] Apache Tomcat: Open Source software implementation of the Java Servlet and Java Server Pages technologies.
http://tomcat.apache.org/

Miscellaneous

[dict01] Developers Dictionary. Definition of 'Model'.
http://dico.developpez.com/html/975-Conception-modele.php

[dict02] Developers Dictionary. Definition of 'Abstraction'.
http://dico.developpez.com/html/974-Generalites-abstraction.php

[dict03] Developers Dictionary. Definition of 'Abstraction'.
http://dico.developpez.com/html/138-Generalites-application.php

[orangepartner] Orange Partner
http://www.orangepartner.com/

7 Author Publications

Scientific papers

1. S. Becot, M. Belaunde, B. Molina: "Empowering Telco Operator Convergence Through a Common Marketplace", ICIN 2010.
2. M.Belaunde, P.Falcarin: "Realizing an MDA and SOA Marriage for the Development of Mobile Services". ECMDA-FA 2008: 393-405
3. O.Droegehorn, I.König, G.Le-Jeune, J.Cupillars, M. Belaunde, E. Kovacs: Professional and end-user-driven service creation in the SPICE platform. WOWMOM 2008: 1-8
4. P.Falcarin, M.Belaunde: First International Workshop on Telecom Service Oriented Architectures (TSOA-07). ICSOC Workshops 2007: 246-247
5. C.Villalonga, M.Strohbach, N.Snoeck, M.Sutterer, M.Belaunde, E.Kovacs, A.Zhdanova, L.W. Goix, O.Droegehorn: "Mobile Ontology: Towards a Standardized Semantic Model for the Mobile Domain". ICSOC Workshops 2007: 248-257
6. María José Presso, Mariano Belaunde: Applying MDA to Voice Applications: An Experience in Building an MDA Tool Chain. ECMDA-FA 2005: 1-8
7. Roy Grønmo, Mariano Belaunde, Jan Øyvind Aagedal, Klaus-D. Engel, Madeleine Faugère, Ida Solheim: Evaluation of the Proposed QVTMerge Language for Model Transformations. WSMDEIS 2005: 65-74
8. Thanh Ha Pham, Mariano Belaunde, Jean Bézivin: Towards a formalization of model conformance in Model Driven Engineering. WSMDEIS 2005: 85-94
9. J.M Jezequel, M. Belaunde, J. Bezivin, S. Gérard et al: OFTA Arago 30 - Rapport de synthèse du Groupe « Ingénierie des modèles » de l'Observatoire Français des Techniques Avancées, Mai 2004. http://ofta.polytechnique.org/Sommaires/30/
10. Anastasius Gavras, Mariano Belaunde, Luís Ferreira Pires, João Paulo A. Almeida: Towards an MDA-Based Development Methodology. EWSA 2004: 230-240
11. M. Belaunde, J.P Almeida, J. Pires, M. Born et al, Modatel Project - "Assessment of the Model Driven Technologies –Foundations and Key Technologies". Section 2.1.4.1. http://www.modatel.org/~Modatel/pub/deliverables/D2.1-final.pdf
12. Mariano Belaunde, Jean Bézivin, Thanh Ha Pham: Implementing EDOC business components on top of a CCM platform. EDOC 2003: 208-221
13. Mariano Belaunde, Mikael Peltier: From EDOC Components to CCM Components: A Precise Mapping Specification. FASE 2002: 143-158
14. María José Presso, Gilbert Raymond, Mariano Belaunde: PILOTE: A Tool Suite to Support UML-Based Engineering Processes. EDOC 2000: 242-251
15. Mariano Belaunde: A Pragmatic Approach for Building a Flexible UML Model Repository. UML 1999 conference: pages 188-203

Cooperative research projects (a selection)

1. TRAMs (RNTL French research project on Information System Modernisation)
2. IST MODA-TEL (Application of MDA to Telecom)
   http://www.modatel.org
3. IST MODELWARE (MDA core technology and Experimentations)
4. OPENEMBEDD (MDA applied to real-time & embedded systems)
   http://openembedd.inria.fr/home_html
5. IST SPICE (Service delivery platform for telecom).
   http://www.ist-spice.org
6. CELTIC SERVERY (Service delivery platform for telecom)
   http://projects.celtic-initiative.org/servery/
7. IST PANLAB (Federation of testbeds)
   http://www.panlab.net

Contributions to industry standards (a selection)

Below we provide a selection of standards in which the author provide significant contribution.

1. SoaML 1.0 [omg-soaml] : Contribution in service composition aspects.
2. QVT 1.0 [omg-qvt]: Chaiman. Main contributor for QVT Operational part.
3. OCL 2.0 [omg-ocl]: Chairman. Contribution to Essential OCL and OCL/UML2 alignment
4. SPEM 1.0 [omg-spem]: Finalisation of the specification.
5. HUTN 1.0 [omg-hutn]: Finalisation of the specification
6. Mda Guide 1.1 [omg-mdag] : Contribution in concepts definition
7. TelcoML: In progress. Adaptation of SoaML to Telecom domain.

8 Annex A: Details of Address Book Experiment

In this annex we provide complementary material regarding the Address Book experiment presented in Section 4.2.1.

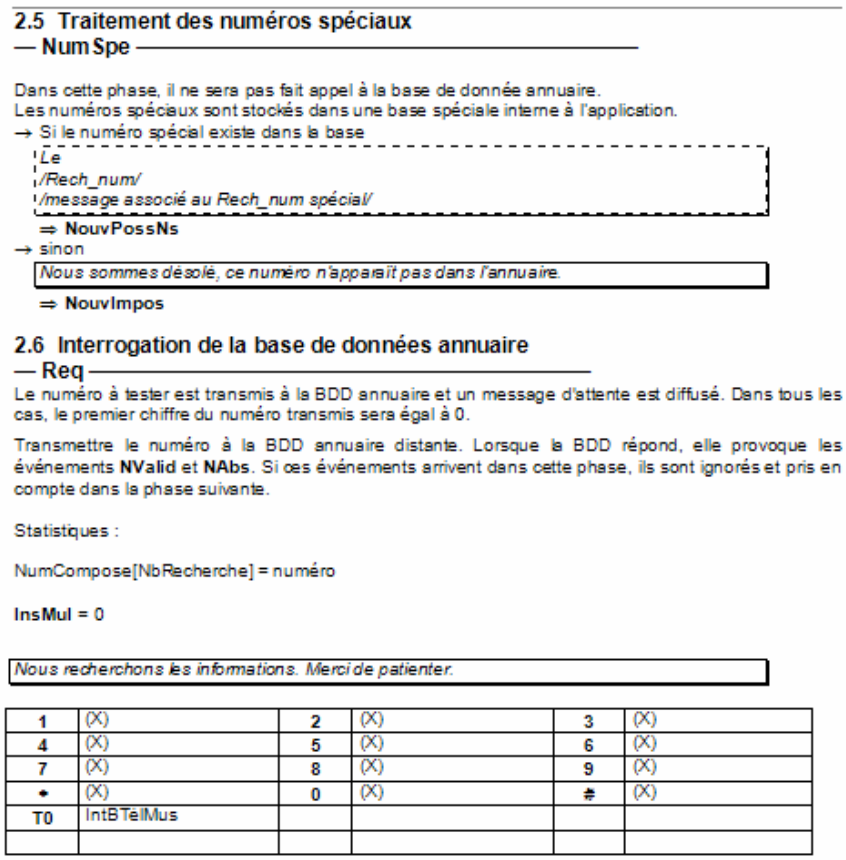
8.1 Realization with Traditional approach

8.1.1 Specification formalism in the traditional approach

The traditional approach for specifying voice dialogs is to use a semi-formal specification written in MS-WORD documents, called DTM7, that merges natural language with pseudo-code. The purpose of this specification is to give all the details that is necessay and reasobnable to implement the state machine of the service.

The Figure below is a screenshot of the specification document – written in French. We see here the specification of the dialog for special reserved phone numbers. A sentence in natural language describes the purpose. The decision logic is written using "if/else" pseudo code ("Si/Sinon" in French). Messages produced are depicted in bordered sentences (with dotted lines for interruptible messages). The invoked sub-dialogs are described in bold and prefixed with '=>' character.

In the bottom of the Figure below we see the transition table for the "query of the address database" dialog ("Interrogation de la base de données annuaire", in French).



2.5 Traitement des numéros spéciaux

— NumSpe —

Dans cette phase, il ne sera pas fait appel à la base de donnée annuaire.
Les numéros spéciaux sont stockés dans une base spéciale interne à l'application.

→ Si le numéro spécial existe dans la base

Le
/Rech_num/
/message associé au Rech_num spécial/

⇒ **NouvPossNs**

→ sinon

Nous sommes désolé, ce numéro n'apparaît pas dans l'annuaire.

⇒ **NouvImpos**

2.6 Interrogation de la base de données annuaire

— Req —

Le numéro à tester est transmis à la BDD annuaire et un message d'attente est diffusé. Dans tous les cas, le premier chiffre du numéro transmis sera égal à 0.

Transmettre le numéro à la BDD annuaire distante. Lorsque la BDD répond, elle provoque les événements **NValid** et **NAbs**. Si ces événements arrivent dans cette phase, ils sont ignorés et pris en compte dans la phase suivante.

Statistiques :

NumCompose[NbRecherche] = numéro

InsMul = 0

Nous recherchons les informations. Merci de patienter.

1	(X)	**2**	(X)	**3**	(X)
4	(X)	**5**	(X)	**6**	(X)
7	(X)	**8**	(X)	**9**	(X)
*****	(X)	**0**	(X)	**#**	(X)
T0	IntBTelMus				

Figure 34: Screenshot of DTMF7 specification

The next Figure show another sub-dialog examples (called phases in the formalism) where we can see the possibility to pass variable parameters. Each phase is described by a list of actions and branching decisions and then by a transition table.

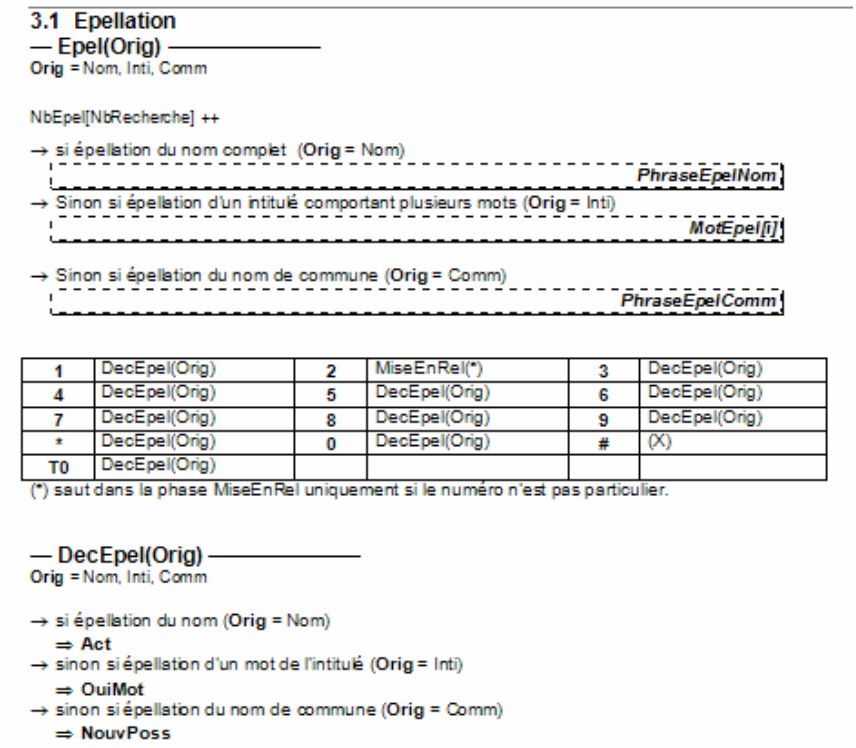


Figure 35: Dialog illustration using parameters

This pseudo-formal representation has the interesting feature to be relatively easy to read and also to be easy to integrate into a formal specification document, serving as a Request for Proposal for potential implementers if the specification.

However the main problems are that the formalism is:

- Difficult to follow and share for non specialist due to the lack of an intuitive graphical notation,
- Contains too many "informal" data which is subjects of various interpretations,
- Cannot be simulated without an important retro-engineering work.
- It is very difficult to generate anything useful for the implementation using this word format.

8.1.2 Implementation of the Address Book service

The Address Book service that we are using as a baseline has been implemented on top of the Euphonie platform as described in Section 3.2 Tools And Processes. In this section we provide some highlights of this implementation.

For each dialog and sub-dialog there is a Java property file which provides the following information:

- The list of states,
- The list of transitions
- The list of recognition orders treated in this section

All this information represents the complete state machine for one dialog.

The Figure below lists all the property files defined for the Address Book service:

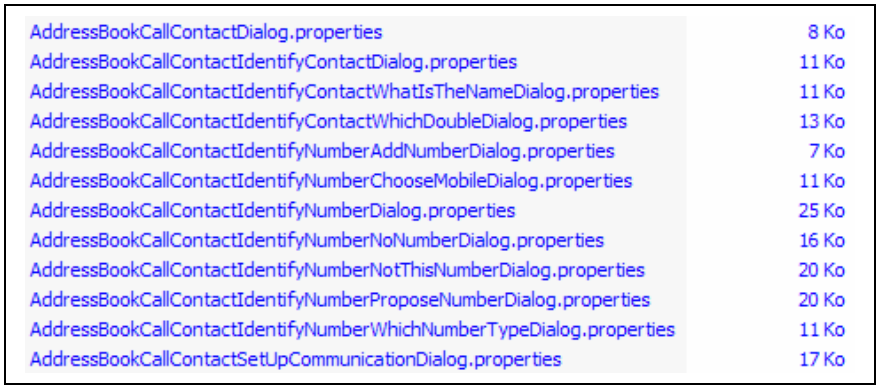


Figure 36: List of property files

The figure below shows some parts of the content of one of these files (the SetUpCommunication module). The first group of assignments defines the attributes of the first state, whereas the second group defines the properties of a transition:

```
# AddressBook_SetUpCommunication_Entry_1
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.State.1.OutGoingTransitions=1
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.State.1.Name=AddressBook_SetUpCommunication_Entry_1
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.State.1.NoInputTimer=0
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.State.1.TemplateName=BlankDialogScreen
```

```
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.Transition.1.RecoOrder=2
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.Transition.1.TargetState=2
WelcomeEngine.Service.AddressBook.CallContactSetUpCommunication
.Transition.1.Type=DialogTransition
```

Figure 37: State and Transition definition

From these examples we can easily understand that the complete coding of the state machine in a Java property file represents a huge work, even if copy-paste editing facilities in the text editor will avoid re-typing the context for each line in the file.

Apart from state-machine definition, the implementer has to implement the code for each condition and each action that are attached to conditional transitions (called contextual controllers) and states (called working states). In the case of the Address Book example, we have 50 ContextController classes and 42 WorkingState classes to develop. The code excerpt below is the code for one specific decision.

```
    public boolean isContextVerified(Order pOrder,Session pSession) {

        //Place isContextVerified's user code here.
        //TODO : Remove this comment if you do any modification here.

        SalientContext lSalientContext = pSession.getWelcomeEngine().getSal
ientContext();
        Object lCurrentObject = pSession.getWelcomeEngine().getSalientConte
xt().remember(WelcomeEngineMemoryConstants.CURRENT_OBJECT);
        HashMap lCurrentObjectsDictionnary = null;
        if (lCurrentObject instanceof HashMap)lCurrentObjectsDictionnary =
(HashMap)lCurrentObject;
        else {
        lCurrentObjectsDictionnary = new HashMap();
        AddressBook addressbook = (AddressBook) pSession.getServiceByClassN
ame(AddressBook.class.getName());
        lCurrentObjectsDictionnary.put("addressbook",addressbook);
        }
          lSalientContext.memorise(WelcomeEngineMemoryConstants.CURRENT_OBJ
ECT,lCurrentObjectsDictionnary);
        return false;

        //end of isContextVerified's user code.
    }
```

Figure 38: Implementing decision code

Finally the implementer provides the code for the Java address book representing the business entities, mainly the address book class which is connected with a database (a MySql Server in our case). The Figure below gives the list of implemented classes with their description.

Class Summary	
AddressBook	This class represents the enter point of the address book service.
AddressBookServerInterface	This class is the interface with the address book server.
ComplementaryInfo	This class represents the complementary information linked up with the Contact class.
Contact	This class represents the java object corresponding to a contact coming from the address book server.
ContactList	This class is a representation of a list of contacts featuring in the address book.
ContactNames	This class is a representation of a contact by means of names

Figure 39: Business entity classes for the Address Book

The Euphonie framework allows developing voice services in a very structured way simplifying the construction of the application. However, due to the absence of automatic generation facilities, the amount of work that the implementer has to provide to obtain a running application is far from being satisfactory. This was one of the motivations for trying to build a MDD tool chain to automate large parts of the implementation.

8.2 Measurements

8.2.1 Measured gain in productivity when using the MDD Voice tool chain

In the previous sections we have discussed mainly the costs for developing the MDD Voice tool chain. We will now look at the productivity gain that such development generates in the work of service designers and services developers.

The strategy for measuring the productivity gain was to take a specific example of a voice service developed in a traditional way and to develop the same service using the voice tool chain: the Address Book voice service presented in Section 4.2.1. To take into account the influence of tool immaturity as well as the influence of pre-existing analysis when re-developing the Address Book service we will apply corrective factors to the obtained measures. However the two measures (observed measures with and without corrective factors) will be kept separately.

8.2.2 Scope and validity of measurements

The quantitative measures are restricted to the case of the Address Book service since it is the service for which we have numbers that can be compared. We believe the measurements done on this example gives a good *estimation* of the productivity increase that can be obtained when intensive code generation techniques are used for developing voice applications. However, we have to be aware that this productivity gain depends a lot on the maturity of the Voice tool chain and on the level of automation provided at some point in its development (the tool is still evolving continuously taking into account the feedback from users). When looking at the measurements we also need to consider the following concerns:

- Number of developed services: Due to reduced period of the experiment it was not possible to develop a much more significant number of large services that would help on tuning the measurements strategy and have a larger number of measures.
- Immaturity of MDD tools: Due to the immaturity of the tools being used it was not always easy to make a good separation between the time spent due to bugs and those that were effectively part of the "regular" design and implementation work.

8.2.3 Effort needed to specify a functional module

For each functional unit, we measure the *time spent* by the designer (in hours, one person per functionality) and we measure the *complexity* of the state machine describing the interaction logic. The effort is defined as a ratio between the time spent and the complexity of the state machine of the functionality.

In the case of the baseline experiment, the state machine is semi-formally described in a word document using an internal proprietary notation (DTMF7). The wait states and

decisions nodes are explicit but some details are simply presented in natural language – like the conditions on decisions. In contrast, in the case of the MDD experiment the state machine is completely formally encoded using the UML case tool (Telelogic TAU 2.3). The measured effort includes the time spent to define the interfaces to access the business entities (the address book application java classes) and the effort to encode decisions and actions in transitions.

Functional units	State Machine Complexity	Baseline experiment (time spent in hours)	MDD experiment (time spent in hours)
IdentifyContact	56	28	22
IdentifyNumber	10	8	4
UpdateContact	12	12	8
ConsultContact	24	16	8
SetUpCommunication	8	6	4

Table 7 : Complexity and resource consumption measures for specifying function units

For each functional units, using the effort formula,

Effort = (Time spent in hours) / (Complexity of the state machine) x 10

We obtain the following effort average:

	Baseline experiment	MDD experiment
Design Effort	7,44	4,56

Table 8 : Average design effort to specify a functional unit

8.2.4 Effort needed to implement a functional module

For each functional unit, we measure the time spent by the developer (one person per functionality) to implement the state machine of the design. This includes the code that makes the link with the pre-existing address book implementation ("glue code") as well as the time needed to encode the grammars.

In the case of the non MDD experiment the state machine is encoded thanks to Java property files. The code implementing the decisions states and the transition actions has also to be written explicitly using specific "behaviour" Java classes.

In the case of the MDD experiment only the body of the operations linking to the business classes (the pre-existing Address Book) need to be provided – the signature of the operations being automatically generated.

Functional units	State Machine Complexity	Baseline experiment (time spent in hours)	MDD experiment (time spent in hours)
IdentifyContact	56	38	10
IdentifyNumber	10	12	5
UpdateContact	12	22	8
ConsultContact	24	32	7
SetUpCommunication	8	10	4

Table 9 : Complexity and resource consumption measures for implementing function units

For all the functional units, using the effort formula (see justification in Deliverable D5.1),

Effort = (Time spent in hours) / (Complexity of the state machine) x 10

We obtain the following average numbers:

	Baseline experiment	MDD experiment
Implementation Effort	12,6	4,3

Table 10 : Average implementation effort to implement a function unit

8.2.5 Corrective factors for design and implementation of a functional module

In order to take into account the fact that the MDD experiment had the advantage of reusing the analysis done in the baseline experiment we will consider a corrective factor of + 30% to the *design* effort measure for the MDD experiment. This factor results from the *estimation* provided by the chief service designer of the MDD variant.

In order to take into account the time spent when facing bugs in the tool chain we will consider a corrective factor of -10% to the *implementation* effort for the MDD experiment. Again this corrective value comes from a global estimation provided by the chief service designer of the MDD variant.

As discussed in the baseline establishment document, it is not possible to evaluate precisely these two factors other than through an approximate *global estimation*: the former factor completely comes from a *subjective* perception of the time saved thanks to a pre-existing knowledge of the application, whereas the second depends on an unrealistic capability to identify, categorize and track every kind of problem encountered during the development of the voice service.

When applying the corrective factors, we have the following numbers:

	Baseline experiment	MDD experiment
Design Effort	7,44	4,56 + 1,36 = 5,92
Implementation Effort	12,6	4,3 – 0,4 = 3,9

Table 11 : Correction factors for effort measurement

8.2.6 Effort needed to change a functional module

We have enriched an existing functional unit – *Identify Contact* – adding the possibility to provide a location keyword (like the country or the city or the street) and measured global cost of this evolution in the design and the implementation. This addition has been done in the baseline application and in the MDD application by the same person. Not that the measure concerns only the enhancements in the *dialog*, not the changes made in the business code (extending the address book data structure with the location keywords) which is shared by the two versions of the application.

The table below provides the measures

	State Machine Complexity	Baseline experiment (time spent in hours)	MDD experiment (time spent in hours)
Identify Contact ++ (design)	12	6,5	4
Identify Contact ++ (impl)	12	12	3

Table 12 : Complexity and resource consumption measures for changing a function unit

And the effort computation

	Baseline experiment	MDD experiment
Design Effort	4,6	3,3
Implementation Effort	10	2,5

Table 13 : Effort to change a function unit

8.2.7 Productivity measure

We define separately the productivity gain for the design and the productivity gain for the implementation. Also we make the distinction when it is first shot or an evolution of the original function. Because of the nature of the corrective factors, which are based on global estimations rather than exact measurements, we will maintain two results: one with the influence of the corrective factors and another without them.

The percentage in productivity gain for each phase is given by the formula:

G = 100 - (MDD x 100 / Baseline)

For the results without the corrective factors we have the following observed productivity gain:

	Productivity Gain
Design (first shot)	+ 39 %
Implementation (first shot)	+ 66 %
Design (evolution)	+ 28 %
Implementation (evolution)	+ 75 %

Table 14 : Productivity gain without corrective factors

If we apply the corrective factors we have the following productivity gain:

	Productivity Gain
Design (first shot)	+20 %
Implementation (first shot)	+69 %
Design (evolution)	+28 %
Implementation (evolution)	+75 %

Table 15 Productivity gain with corrective factors

Unsurprisingly we note that the productivity gain is very high for the implementation since most of it is generated automatically when using an MDD approach. Design productivity gain is affected by the fact that much more effort has to be put in this phase in order to allow high automation whereas in a non MDD approach this phase may leave imprecise some parts of the specification.

For France Telecom we should point out that productivity in the design phase is much more important than design in implementation since approx. only ¼ of the voice services are effectively implemented by France Telecom whereas the others are only designed and left to third parties for the implementation. Note that the measured design productivity does not take into account the simulation aspect which may influence the final cost of a service.

The following table gives an average of the 4 numbers of productivity gains obtained separately for each categories (design/implementation and first short/evolution).

	Average of Productivity Gain
Without corrective factors	+52%
With corrective factors	+48%

Table 16 : Average of productivity gain

It is important to remind that this productivity gain is the observed productivity gain for one voice service (The Address Book) service and that it does not includes the cost for building and maintaining the MDD voice tool chain (which is mutualised by all services that can potentially be developed with it).

9 Annex B: SPATEL Technical Artefacts

In this annex we provide the complete definition of some technical artefacts.

9.1 SPATEL metamodel

The complete SPATEL metamodel defined using EMOF textual syntax (defined in the OMG QVT specification) is provided below.

```
// ######################################################
// #  METAMODEL spatel
// #   (generated by PyMof)
// ######################################################

package spatel {

  // type definitions
  class AcceptEventAction extends Action {
    composes argument : OrderedSet(OclExpression); // [*]
    event : ServiceEvent; // [1]
  }

  abstract class Action extends Element {
    opaqueBody : String;
  }

  class ActionSequence extends Action {
    name : String;
    composes action : OrderedSet(Action); // [*]
  }

  class AssignmentAction extends Action {
    composes left : OclExpression; // [1]
    composes right : OclExpression; // [0..1]
  }

  class CallAction extends ExpressionAction {
  }

  abstract class ExpressionAction extends Action {
    composes expression : OclExpression; // [1]
  }

  class InformalExp extends OclExpression {
    body : String;
  }

  class NewExp extends OclExpression {
    composes argument : OrderedSet(OclExpression); // [*]
    targetType : Class; // [1]
  }

  class SendEventAction extends Action {
    composes argument : OrderedSet(OclExpression); // [*]
    event : ServiceEvent; // [1]
  }

  class UninterpretedAction extends Action {
    body : String;
  }

  class Class extends Type {
```

```
  isAbstract : Boolean;
  composes ownedAttribute : OrderedSet(Property); // [*]
  composes ownedOperation : OrderedSet(Operation); // [*]
  superClass : OrderedSet(Class); // [*]
}

class Comment extends Element {
  body : String;
}

class DataType extends Type {
}

abstract class Element {
  composes ownedTag : OrderedSet(Tag); // [*]
  composes ownedComment : OrderedSet(Comment); // [*]
}

class Enumeration extends Type {
  composes ownedLiteral : OrderedSet(EnumerationLiteral); // [*]
}

class EnumerationLiteral extends NamedElement {
}

abstract class MultiplicityElement extends Element {
  isOrdered : Boolean;
  isUnique_ : Boolean;
  lower : Integer;
  upper : UnlimitedNatural;
}

abstract class NamedElement extends Element {
  name : String;
}

class Operation extends MultiplicityElement,TypedElement {
  composes ownedParameter : OrderedSet(Parameter); // [*]
  composes raisedException : OrderedSet(Type); // [*]
}

class Package extends NamedElement {
  uri : String;
  composes ownedType : OrderedSet(Type); // [*]
  composes nestedPackage : OrderedSet(Package); // [*]
}

class Parameter extends MultiplicityElement,TypedElement {
}

class PrimitiveType extends DataType {
}

class Property extends MultiplicityElement,TypedElement,Type {
  isReadOnly : Boolean;
  isDerived : Boolean;
  isId : Boolean;
  default_ : String;
  composes opposite : Property; // [0..1]
}

class Tag {
  value : String;
  name : String;
  element : OrderedSet(Element); // [*]
}

abstract class Type extends NamedElement {
}

abstract class TypedElement extends NamedElement {
  type : Type; // [0..1]
```

```
}

class PlayAction extends Action {
  isInterruptible : Boolean;
  composes argument : OrderedSet(OclExpression); // [*]
  message : VoiceMessage; // [1]
}

class VoiceMessage extends ServiceElement {
  diffusionMode : String;
  text : String;
  composes ownedPart : OrderedSet(VoiceMessage); // [*]
  composes parameter : OrderedSet(Parameter); // [*]
  composes bodyExpression : OclExpression; // [0..1]
  usedPart : OrderedSet(VoiceMessage); // [*]
}

class DtmfEvent extends ServiceEvent {
  key_ : String;
}

class SystemEvent extends ServiceEvent {
}

class DiversionState extends FinalState {
  called : Dialog; // [1]
}

class Dialog extends ServiceComponent {
  composes message : OrderedSet(VoiceMessage); // [*]
}

class SubDialogState extends State {
  called : Dialog; // [1]
}

class RecoEvent extends ServiceEvent {
}

class AnyType extends Class,Type {
}

class BagType extends CollectionType {
}

class BooleanLiteralExp extends PrimitiveLiteralExp {
  booleanSymbol : Boolean;
}

abstract class CallExp extends OclExpression {
  composes source : OclExpression; // [0..1]
}

class CollectionItem extends CollectionLiteralPart {
  composes item : OclExpression; // [1]
}

class CollectionLiteralExp extends LiteralExp {
  kind : CollectionKind;
  composes part : OrderedSet(CollectionLiteralPart)
      opposites CollectionLiteralExp; // [*],[1]
}

abstract class CollectionLiteralPart extends TypedElement {
}

class CollectionRange extends CollectionLiteralPart {
  composes first : OclExpression; // [1]
  composes last : OclExpression; // [1]
}

abstract class CollectionType extends DataType {
```

```
  elementType : Type; // [0..1]
}

class EnumLiteralExp extends LiteralExp {
  referredEnumLiteral : EnumerationLiteral; // [0..1]
}

class ExpressionInOcl {
  composes bodyExpression : OclExpression; // [1]
  composes context : Variable; // [0..1]
  composes resultVariable : Variable; // [0..1]
  composes parameterVariable : Variable; // [0..1]
}

abstract class FeaturePropertyCall extends CallExp {
}

class IfExp extends OclExpression {
  condition : OclExpression; // [1]
  thenExpression : OclExpression; // [1]
  elseExpression : OclExpression; // [1]
}

class IntegerLiteralExp extends NumericLiteralExp {
  integerSymbol : Integer;
}

class InvalidLiteralExp extends LiteralExp {
}

class InvalidType extends Type {
}

class IterateExp extends LoopExp {
  result : Variable; // [0..1]
}

class IteratorExp extends LoopExp {
}

class LetExp extends OclExpression {
  composes variable : Variable; // [1]
  in_ : OclExpression; // [1]
}

abstract class LiteralExp extends OclExpression {
}

abstract class LoopExp extends CallExp,OclExpression {
  composes iterator : OrderedSet(Variable); // [*]
  body : OclExpression; // [1]
}

class NullLiteralExp extends LiteralExp {
}

abstract class NumericLiteralExp extends PrimitiveLiteralExp {
}

abstract class OclExpression extends TypedElement {
}

class OperationCallExp extends FeaturePropertyCall {
  composes argument : OrderedSet(OclExpression); // [*]
  referredOperation : Operation; // [0..1]
}

class OrderedSetType extends CollectionType {
}

abstract class PrimitiveLiteralExp extends LiteralExp {
}
```

```
class PropertyCallExp extends FeaturePropertyCall {
  referredProperty : Property; // [0..1]
}

class RealLiteralExp extends NumericLiteralExp {
  realSymbol : Real;
}

class SequenceType extends CollectionType {
}

class SetType extends CollectionType {
}

class StringLiteralExp extends PrimitiveLiteralExp {
  stringSymbol : String;
}

class TupleLiteralExp extends LiteralExp {
  composes part : OrderedSet(TupleLiteralPart)
      opposites TupleLiteralExp; // [*],[0..1]
}

class TupleLiteralPart extends TypedElement {
  composes attribute : Property; // [0..1]
  composes value : OclExpression; // [1]
}

class TupleType extends Class,DataType {
}

class TypeExp extends OclExpression {
  referredType : Type; // [0..1]
}

class UnlimitedNaturalExp extends NumericLiteralExp {
  symbol : UnlimitedNatural;
}

class Variable extends TypedElement {
  bindParameter : Parameter; // [0..1]
  initExpression : OclExpression; // [0..1]
}

class VariableExp extends OclExpression {
  referredVariable : Variable; // [1]
}

class VoidType extends Type {
}

class AnyReceivedEvent extends ServiceEvent {
}

class ServiceChangeEvent extends ServiceEvent {
  composes changeExpression : OclExpression; // [1]
}

class ServiceTimeEvent extends ServiceEvent {
  isRelative : Boolean;
  composes when_ : OclExpression; // [1]
}

class NonFunctionalTag extends ServiceElement {
  category : String;
  value : String;
  isDynamic : Boolean;
  criterion : String;
}

class OntologyUsage extends ServiceElement {
```

```
  uri : String;
}

class SemanticTag extends ServiceElement {
  kind : String;
  value : String;
}

class ServiceAttribute extends Property,ServiceElement {
  kind : String;
  instanceType : String;
}

abstract class ServiceBehavior extends Class,ServiceElement {
}

class ServiceCollaboration extends ServiceElement {
  composes interaction : OrderedSet(ServiceInteraction); // [*]
}

class ServiceComponent extends ServiceElement,ServiceNamespace {
  kind : String;
  isPureContainer : Boolean;
  isFlat : Boolean;
  composes interactionPoint : OrderedSet(ServicePort)
      opposites owner; // [*],[0..1]
  composes method : OrderedSet(ServiceMethod)
      opposites owner; // [*],[0..1]
  composes usedComponent : OrderedSet(ServiceComponent)
      opposites parent; // [*],[0..1]
  composes collaboration : OrderedSet(ServiceCollaboration)
      opposites component; // [*],[0..1]
  composes ServiceOperation : OrderedSet(ServiceOperation); // [*]
  composes data : OrderedSet(ServiceAttribute)
      opposites componentOwner; // [*],[0..1]
  composes interactionInterface : OrderedSet(ServiceInteractionInterface); // [*]
  typeInterface : ServiceInterface; // [1]
  additionalInterface : OrderedSet(ServiceInterface); // [*]
  representedComponent : ServiceComponent; // [0..1]
}

class ServiceConnection extends ServiceElement {
  isDelegation : Boolean;
  targetPort : OrderedSet(ServicePort)
      opposites incomingConnection; // [*],[*]
}

class ServiceContract extends ServiceElement {
  composes definition : ServiceBehavior
      opposites contractOwner; // [0..1],[0..1]
}

abstract class ServiceElement extends NamedElement {
  semType : String;
  semPattern : String;
  composes semTag : OrderedSet(SemanticTag)
      opposites owner; // [*],[0..1]
  composes nonFuncTag : OrderedSet(NonFunctionalTag)
      opposites owner; // [*],[0..1]
}

class ServiceEntity extends Class,ServiceElement {
  scope : String;
  kind : String;
  representedType : Type; // [0..1]
}

class ServiceEvent extends Class,ServiceElement {
  kind : String;
  composes parameter : OrderedSet(ServiceParameter)
      opposites ownerEvent; // [*],[0..1]
}
```

```
class ServiceException extends Class,ServiceElement {
}

class ServiceInteraction extends ServiceElement {
  kind : String;
  source : ServiceElement; // [1]
  target : ServiceElement; // [1]
  connection : ServiceConnection; // [1]
}

class ServiceInteractionInterface extends ServiceInterface {
}

class ServiceInterface extends Class,ServiceElement {
  isOrchestration : Boolean;
  composes ontology : OrderedSet(OntologyUsage)
      opposites ServiceInterface; // [*],[0..1]
  composes contract : ServiceContract
      opposites interface; // [0..1],[0..1]
  composes ui : UiContainer; // [0..1]
  defaultOperation : ServiceOperation; // [0..1]
  generatedEvent : OrderedSet(ServiceEvent); // [*]
  acceptedEvent : OrderedSet(ServiceEvent); // [*]
  sentEvent : OrderedSet(ServiceEvent); // [*]
}

class ServiceLibrary extends ServiceElement,Package,ServiceNamespace {
  composes service : OrderedSet(ServiceInterface); // [*]
  composes serviceComponent : OrderedSet(ServiceComponent); // [*]
  composes client : OrderedSet(ServiceClient); // [*]
  composes ui : UiContainer; // [0..1]
  mainServicePackage : ServicePackage; // [0..1]
  mainService : ServiceInterface; // [0..1]
}

class ServiceMethod extends Operation,ServiceElement {
  isOpaque : Boolean;
  composes behavior : ServiceBehavior
      opposites realizationOwner; // [0..1],[0..1]
  specification : ServiceOperation; // [0..1]
}

abstract class ServiceNamespace {
  composes event : OrderedSet(ServiceEvent)
      opposites namespace; // [*],[0..1]
  composes stream : OrderedSet(ServiceStream); // [*]
  composes sideEffect : OrderedSet(ServiceSideEffect); // [*]
}

class ServiceOperation extends Operation,ServiceElement {
  kind : String;
  composes behavior : ServiceBehavior; // [0..1]
  sentEvent : OrderedSet(ServiceEvent); // [*]
  acceptedEvent : OrderedSet(ServiceEvent); // [*]
  triggeredBy : OrderedSet(ServiceEvent); // [*]
  sideEffect : OrderedSet(ServiceSideEffect); // [*]
  outStream : OrderedSet(ServiceStream); // [*]
  inStream : OrderedSet(ServiceStream); // [*]
}

class ServiceParameter extends Parameter,ServiceElement {
  direction : String;
  instanceType : String;
}

class ServicePort extends ServiceElement {
  direction : String;
  composes outgoingConnection : OrderedSet(ServiceConnection)
      opposites sourcePort; // [*],[0..1]
  representedElement : ServiceElement; // [0..1]
  incomingConnection : OrderedSet(ServiceConnection)
```

```
      opposites targetPort; // [*],[*]
}

class ServiceSideEffect extends ServiceElement {
  kind : String;
}

class ServiceStream extends ServiceElement {
}

class FinalState extends State {
}

class Guard extends NamedElement {
  opaqueBody : String;
  composes expression : OclExpression; // [0..1]
}

abstract class Pseudostate extends Vertex {
  kind : String;
}

class Region extends NamedElement {
  composes subvertex : OrderedSet(Vertex)
      opposites owner; // [*],[0..1]
  composes transition : OrderedSet(Transition)
      opposites owner; // [*],[0..1]
}

abstract class State extends Vertex {
  composes deferrableTrigger : OrderedSet(Trigger)
      opposites state; // [*],[0..1]
}

class StateMachine extends ServiceBehavior {
  composes region : OrderedSet(Region)
      opposites owner; // [*],[0..1]
  composes variable : OrderedSet(Variable); // [*]
  composes ui : UiContainer; // [0..1]
  composes initSection : ActionSequence; // [0..1]
  composes endSection : ActionSequence; // [0..1]
}

class Transition extends NamedElement {
  isElse : Boolean;
  composes trigger : OrderedSet(Trigger)
      opposites transition; // [*],[0..1]
  composes effect : ActionSequence
      opposites transition; // [0..1],[0..1]
  composes guard : Guard; // [0..1]
  source : Vertex
      opposites outgoing; // [1],[*]
  target : Vertex
      opposites incoming; // [1],[*]
}

class Trigger extends NamedElement {
  composes event : ServiceEvent; // [1]
  composes acceptAction : OrderedSet(AcceptEventAction); // [*]
  composes filter : OclExpression; // [0..1]
}

abstract class Vertex extends NamedElement {
  composes entryAction : ActionSequence; // [0..1]
  composes exitAction : ActionSequence; // [0..1]
  composes activationGuard : OclExpression; // [0..1]
  composes variable : OrderedSet(Variable); // [*]
  outgoing : OrderedSet(Transition)
      opposites source; // [*],[1]
  incoming : OrderedSet(Transition)
      opposites target; // [*],[1]
}
```

```
class UiElement extends Variable {
  kind : String;
  composes attribute : OrderedSet(UiProperty)
      opposites uiElement; // [*],[1]
  composes ownedEvent : OrderedSet(UiEvent)
      opposites uiOwner; // [*],[1]
  composes trigger : UiTrigger; // [0..1]
}

class UiContainer extends UiElement {
  composes element : OrderedSet(UiElement)
      opposites container; // [*],[1]
}

class UiProperty extends Variable {
  value : String;
  linkValue : UiElement; // [0..1]
}

class UiEvent extends ServiceEvent {
  bindExp : String;
  bindTo : ServiceEvent; // [0..1]
}

class ServicePackage extends ServiceLibrary {
}

class ServiceClient extends ServiceElement,Package {
  composes ui : UiContainer; // [0..1]
}

class UiTrigger extends Trigger {
  composes effect : ActionSequence
      opposites uiTrigger; // [0..1],[0..1]
}

class InitialNode extends Pseudostate {
}

class LabelNode extends Pseudostate {
}

class JumpNode extends Pseudostate {
  label : LabelNode; // [1]
}

class HistoryState extends Pseudostate {
}

class DeepHistoryState extends Pseudostate {
}

class JoinState extends Pseudostate {
}

class ForkState extends Pseudostate {
}

class JunctionState extends Pseudostate {
}

class ChoiceNode extends Pseudostate {
}

class TerminateNode extends Pseudostate {
}

class WaitState extends State {
}

class ActionSequenceNode extends Pseudostate {
```

```
    composes actionSequence : ActionSequence; // [1]
  }

  class SyncCallState extends CallState {
  }

  class SendNode extends Pseudostate {
    composes eventAction : SendEventAction; // [1]
  }

  class AcceptNode extends Pseudostate {
    composes trigger : Trigger; // [1]
  }

  class RestartNode extends Pseudostate {
  }

  class AsyncCallState extends CallState {
  }

  class SubMachineState extends CallState {
  }

  abstract class CallState extends State {
    composes callAction : Action; // [1]
  }

  class VariableDeclarationAction extends AssignmentAction {
  }

  class ServiceOrchestrationPackage extends ServicePackage {
  }

  class SystemVariable extends Variable {
    role : String;
    scope : String;
  }

  // aliases used within this metamodel
  tag 'alias' MultiplicityElement::isUnique_ = 'isUnique';
  tag 'alias' Property::default_ = 'default';
  tag 'alias' DtmfEvent::key_ = 'key';
  tag 'alias' LetExp::in_ = 'in';
  tag 'alias' ServiceTimeEvent::when_ = 'when';
}
```

9.2 SPATEL Textual Grammar

In this section we provide the grammar of SPATEL textual notation using a specifc lex/yacc notation used by PLY tool (http:// *www.dabeaz.com/**ply**/)*

```
## SPATEL TEXTUAL GRAMMAR

    toplevel : module_element_list_opt
    module_element_list_opt : module_element_list
                            | empty
    module_element : package
                           | classifier
                           | deployment
                           | behavior

    deployment : DEPLOYMENT ID LBRACE deployment_element_list_opt RBRACE
    deployment_element_list_opt : deployment_element_list
                                | empty
    deployment_element_list : deployment_element
                                | deployment_element_list deployment_element
    deployment_element : deployment_service
    deployment_service : SERVICE ID LBRACE deployment_service_element_list_opt RBRACE
    deployment_service_element_list_opt : deployment_service_element_list
```

```
                            | empty
    deployment_service_element_list : deployment_service_element
                            | deployment_service_element_list deployment_service_element
    deployment_service_element : attribute_value_decl
                                    | map
                                    | namespace
    map : MAP ID AS mapped_operation_signature SEMI
    namespace : NAMESPACE attribute_value_comma_list_opt SEMI
    attribute_value_decl : attribute_value SEMI
    attribute_value_comma_list_opt : attribute_value_comma_list
                            | empty
    attribute_value_comma_list : attribute_value
                            | attribute_value_comma_list COMMA attribute_value
    attribute_value : ID EQUALS literal
    mapped_operation_signature : mapped_id LPAREN mapped_parameter_list_opt RPAREN COLON
mapped_parameter_list
    mapped_id : ID
                | ID ARROBAS ID
    mapped_parameter_list_opt : mapped_parameter_list
                            | empty
    mapped_parameter_list : mapped_parameter
                      | mapped_parameter_list COMMA mapped_parameter
    mapped_parameter : mapped_id COLON mapped_scoped_id
                | direction_kind mapped_id COLON mapped_scoped_id
                |  mapped_scoped_id
                | direction_kind mapped_scoped_id
    mapped_scoped_id : mapped_id
                | mapped_scoped_id DCOLON mapped_id

    package : package_kind ID LBRACE package_element_list_opt RBRACE
    package_kind : SERVICELIBRARY
                   | SERVICEPACKAGE
    package_element_list_opt : package_element_list
                           | empty
    package_element_list : package_element
           | package_element_list package_element
    package_element : classifier
                     | behavior
    classifier_kind : SERVICE
                       | EVENT
                       | ENTITY
                       | DATATYPE
    classifier : classifier_kind ID LBRACE classifier_element_list_opt RBRACE
    classifier_element_list_opt : classifier_element_list
                            | classifier_element_list SEMI
                            | empty
    classifier_element_list : classifier_element
                            | classifier_element_list SEMI classifier_element
    classifier_element : operation
               | attribute
               | behavior
    operation : OPERATION operation_signature
    operation_signature : scoped_id LPAREN parameter_list_opt RPAREN
                          | scoped_id LPAREN parameter_list_opt RPAREN COLON parameter_list
    parameter_list_opt : parameter_list
                            | empty
    parameter_list : parameter
                      | parameter_list COMMA parameter
    parameter : ID COLON scoped_id
                | direction_kind ID COLON scoped_id
                | scoped_id
                | direction_kind scoped_id
    direction_kind : IN
                      | INOUT
    scoped_id : ID
                | scoped_id DCOLON ID
    attribute :  ID COLON scoped_id
                | ID COLON scoped_id EQUALS literal
    behavior : BEHAVIOR operation_signature LBRACE behavior_element_list_opt RBRACE
    behavior_element_list_opt : behavior_element_list
                            | empty
    behavior_element_list : behavior_element
```

```
                            | behavior_element_list behavior_element
    behavior_element : node
    node : ID id_expr LBRACE node_element_list_opt RBRACE
    id_expr : attribute_value
                | expression
    node_element_list_opt : node_element_list
                            | node_element_list SEMI
                            | empty
    node_element_list : node_element
                            | node_element_list SEMI node_element
    node_element : action
                   | transition
    action : variable
              | assignment
              | expression
    op_assignment : EQUALS
                     | XEQUALS
                     | PLUSEQUAL
                     | MINUSEQUAL
    assignment : postfix_expr op_assignment expression
    variable : VAR ID COLON scoped_id op_assignment expression
              | VAR ID COLON scoped_id
              | VAR ID op_assignment expression
              | VAR ID
              | USES ID COLON scoped_id
    transition : TRANSITION ARROW expression

## expressions
    literal : integer_literal
              |  float_literal
              |  string_literal
              |  boolean_literal
              |  null_literal
    integer_literal : ICONST
    float_literal : FCONST
    boolean_literal : TRUE
                     | FALSE
    null_literal : NULL
    string_literal : CCONST
                      | SCONST
    arg_list_opt : arg_list
                   | empty
    arg_list : expression
                | arg_list COMMA expression
    unary_op : MINUS
                | NOT
                | INFORMAL
                | NEW
    access_op : PERIOD
                 | ARROW
    logic_and_op : AND
    logic_or_op : OR
              | XOR
    cmp_op : EQ
                | NE
                | NEX
                | LT
                | GT
                | LE
                | GE
    add_op : PLUS
                | MINUS
    mult_op : TIMES
                | DIVIDE
                | MOD
    expression : or_expr
    or_expr : and_expr
               | or_expr logic_or_op and_expr
    and_expr : cmp_expr
                | and_expr logic_and_op cmp_expr
    cmp_expr : additive_expr
                | cmp_expr cmp_op additive_expr
```

```
    additive_expr : mult_expr
                  | additive_expr add_op mult_expr
    mult_expr : unary_expr
              | mult_expr mult_op unary_expr
    unary_expr : postfix_expr
               | unary_op unary_expr
    postfix_expr : primary_expr
                 | postfix_expr LBRACKET expression RBRACKET
                 | postfix_expr LPAREN arg_list_opt RPAREN
                 | postfix_expr access_op ID

   primary_expr : literal
                | scoped_id
                | LPAREN expression RPAREN


   empty :
```

9.3 SPATEL to WSDL Transformation

In this section we provide the complete definition of one important transformation in SPATEL Engine implemented using the QVT Operational Transformation language. This transformation allows the publication of SPATEL definitions in the form of WSDL files.

```
transformation Wsdl2Spatel(in wsdlmodel:WSDL,out spatelmodel:SPATEL);

main() {
  wsdlmodel->objectsOfType(WSDL::Description)->map toServiceLibrary();
  log("Number of root objects: ",spatelmodel.rootObjects()->size());
  log("Number of created objects: ",spatelmodel.objects()->size());
}

query WSDL::Operation::requiresServiceOperation() : Boolean {
  // return self.pattern<>WSDL::Pattern::out_only
  //    and self.pattern<>WSDL::Pattern::robust_out_only;
  return true; // to be refined
}

query WSDL::Operation::requiresServiceEvent() : Boolean {
  // return self.pattern==WSDL::Pattern::out_only
  //       or self.pattern==WSDL::Pattern::robust_out_only;
  return false;
}


mapping WSDL::Description::toServiceLibrary() : SPATEL::ServiceLibrary {
  name := "Service Design";
  uri := self.targetNameSpace;
  ownedComment := if (self.documentation<>null)
     object Comment {
        body := "documentation: "+self.documentation;}
     endif;
  nestedPackage := self.service->map toServicePackage();
}

mapping WSDL::Service::toServicePackage() : SPATEL::ServicePackage {
  name := self.name;
  ownedType := self.container().oclAsType(Description).schema->map toDataTypes();
  ownedType += self.interface.fault->map toServiceException();
  service := self.interface.map toServiceInterface(result);
}

mapping WSDL::Interface::toServiceInterface(packOwner:SPATEL::ServicePackage)
  : SPATEL::ServiceInterface {
  name := self.name;
  ownedOperation := self.operation->map toServiceOperation();
  acceptedEvent :=  self.operation->map toServiceEvent();
  // Attaching new events to the parent package as owned event types
```

```
  end {packOwner.event += result.acceptedEvent;}
}

mapping WSDL::Fault::toServiceException() : SPATEL::ServiceException {
  name := self.name;
}

mapping WSDL::Operation::toServiceOperation() : SPATEL::ServiceOperation
  when {self.requiresServiceOperation()} {
  name := self.name;
  ownedParameter := {
      self.input->map toServiceParameter();
      self.output->map toServiceParameter();
      };
  raisedException := self.outFault->fault->resolveone(SPATEL::ServiceException);
}

mapping WSDL::Operation::toServiceEvent() : SPATEL::ServiceEvent
  when {self.requiresServiceEvent()} {
  name := self.name;
  ownedAttribute := self.output->map toServiceAttribute();
}

mapping WSDL::Input::toServiceParameter()  : SPATEL::ServiceParameter {
  name := self.messageLabel;
  direction := "in";
}

mapping WSDL::Output::toServiceParameter()  : SPATEL::ServiceParameter {
  name := self.messageLabel;
  direction := "out";
}

mapping WSDL::Output::toServiceAttribute()  : SPATEL::ServiceAttribute {
  name := self.messageLabel;
}

mapping WSDL::XSDSchema::toDataTypes() : Sequence(SPATEL::Type) {
  init {
     result := self.element->map toAnyType();
  }
}

mapping WSDL::XSDElement::toAnyType() : SPATEL::Type {
  init {
    result := if (self.complexType==null) self.map toSimpleDataType()
              else self.map toComplexDataType();
  }
}

mapping WSDL::XSDElement::toSimpleDataType() : SPATEL::DataType {
  name := self.name;
}

mapping WSDL::XSDElement::toComplexDataType() : SPATEL::ServiceEntity {
  name := self.name;
  ownedAttribute := self.complexType.sequence->
    collect(i|object ServiceAttribute{name:=i.name;});
}
```

9.4 Generation of a Service

In this section we provide an example of artefacts produced from a SPATEL definition of a basic service. The example is a Translation service interface (no explicit logic in SPATEL is provided for the operation). We show the original specification in textual and XMI form, then the generated python code and the generated WSDL for publication of the service as a SOAP service.

9.4.1 The original SPATEL source file in textual format

```
service Translation {
  operation translate(text:String,sourceLanguage:String,targetLanguage:String) : String;

}
```

9.4.2 The corresponding SPATEL XMI source file

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:spatel="http://istspice.org/spatel/1.0.0/Spatel">
<spatel:ServicePackage xmi:id="o0" name="Translation">
  <service xmi:id="o1" xsi:type="spatel:ServiceInterface" name="Translation">
    <ownedOperation xmi:id="o2" xsi:type="spatel:ServiceOperation" name="translate">
      <ownedParameter xmi:id="o3" xsi:type="spatel:ServiceParameter" direction="in"
instanceType="String" name="text"/>
      <ownedParameter xmi:id="o4" xsi:type="spatel:ServiceParameter" direction="in"
instanceType="String" name="sourceLanguage"/>
      <ownedParameter xmi:id="o5" xsi:type="spatel:ServiceParameter" direction="in"
instanceType="String" name="targetLanguage"/>
      <ownedParameter xmi:id="o6" xsi:type="spatel:ServiceParameter" direction="return"
instanceType="String" name=""/>
    </ownedOperation>
  </service>
</spatel:ServicePackage>
</xmi:XMI>
```

9.4.3 The generated python skeleton code

```
import sys, os

from CONF_ENT_ABSTRACTCATALOG_TRANSLATION import PLUGINCONF
from voicebench.engine.Framework import VBEntity, SpatelSystem

def invokeWithInterpreter(opname,params,session):
    opId = '%s.%s::%s' % (PLUGINCONF.PLUGINID,'Translation',opname)
    from appabstractcatalog.utils.InterpreterHandler import InterpreterHandler
    interpreter = InterpreterHandler(opId,params,session=session)
    return interpreter.resolveAndInvoke()

class TranslationError(Exception): pass

class Translation (VBEntity):
    ## meta information
    META = {
        'translate' : {'args' : ('text', 'sourceLanguage', 'targetLanguage'),},
    }

    def __init__(self,SESSION):
        self.SESSION = SESSION
        self.appId = PLUGINCONF.PLUGINID
        self.spatelsystem = SpatelSystem(
          SESSION,self.appId,"Translation")

    ## *** operation translate ***

    def translate(self,text,sourceLanguage,targetLanguage):
        ## in text:String,in sourceLanguage:String,in targetLanguage:String -> String
        from voicebench.comm.VariantManager import invokeVariant
        return invokeVariant(self,'translate',text,sourceLanguage,targetLanguage)

    def translate_v0(self,text,sourceLanguage,targetLanguage):
        ## in text:String,in sourceLanguage:String,in targetLanguage:String -> String
        ## use this for fake implementation
        result = "" ## default result
        return result

    def translate_v1(self,text,sourceLanguage,targetLanguage):
```

```
        ## in text:String,in sourceLanguage:String,in targetLanguage:String -> String
        ## use this for local implementation
        params =
{'text':text,'sourceLanguage':sourceLanguage,'targetLanguage':targetLanguage}
        result = invokeWithInterpreter('translate',params,session=self.SESSION)
        return result

    def translate_v2(self,text,sourceLanguage,targetLanguage):
        ## in text:String,in sourceLanguage:String,in targetLanguage:String -> String
        ## default remote implementation
        from voicebench.comm.SoapSupport import invokeService
        result = invokeService(
            self,'Translation::translate',
            [text,sourceLanguage,targetLanguage])
        return result

from voicebench.comm.batchlauncher import testEntity

## main
if __name__=="__main__":
    testEntity(Translation)
```

9.4.4 The generated WSDL file

```
<?xml version="1.0" encoding="UTF-8"?>
<!--WSDL created by Spatel Studio -->
<wsdl:definitions name="Translation" targetNamespace="http://istspice.org/wsdl/Translation"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:impl="http://istspice.org/wsdl/Translation">
  <wsdl:types>
    <schema elementFormDefault="qualified"
targetNamespace="http://istspice.org/wsdl/Translation"
xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="translate">
        <complexType>
          <sequence>
            <element name="text" type="xsd:string"/>
            <element name="sourceLanguage" type="xsd:string"/>
            <element name="targetLanguage" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
      <element name="translateResponse">
        <complexType>
          <sequence>
            <element name="translateReturn" type="xsd:string"/>
          </sequence>
        </complexType>
      </element>
    </schema>
  </wsdl:types>
  <wsdl:message name="translateRequest">
    <wsdl:part element="impl:translate" name="parameters"/>
  </wsdl:message>
  <wsdl:message name="translateResponse">
    <wsdl:part element="impl:translateResponse" name="parameters"/>
  </wsdl:message>
  <wsdl:portType name="Translation">
    <wsdl:operation name="translate">
      <wsdl:input name="translateRequest" message="impl:translateRequest"/>
      <wsdl:output name="translateResponse" message="impl:translateResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="TranslationSoapBinding" type="impl:Translation">
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="translate">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="translateRequest">
        <wsdlsoap:body use="literal"/>
      </wsdl:input>
```

```
      <wsdl:output name="translateResponse">
        <wsdlsoap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Translation">
    <wsdl:port name="Translation" binding="impl:TranslationSoapBinding">
      <wsdlsoap:address location="http://localhost/natmashups-site/cgi-
bin/natportal.persoservices.abstractcatalog.ws_translation.py" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

10 Annex C: Natural Mashups Experiment

In this section we provide a brief report on an ongoing experiment to create *on the fly* service compositions as result of the interpretation of a service requests expressed using a simplified form of *natural language*. Since the tooling exploits SPATEL language and the SPATEL Engine infrastructure as the default framework to execute and manage the generated service compositions, this experiment represents an interesting use case of the methods and design principles presented in Section 3.1 and Section 3.2. In particular, the use of a neutral high-level formalism to express orchestrations (SPATEL in our case) ensures decoupling between the natural-language interpretation system creating the compositions and the environment used to execute them. In addition, the variability mechanism offered by the SPATEL Engine framework is used here to implement context-aware abstract services as explained below.

The tool is provided in the form of a web application (see Figure below) that sequences four or five actions: (i) it receives the input request, (ii) interprets the request, (iii) generates the orchestration script that fits with the request, (iv) executes the composite service with available data, or asks for arguments values if some are missing (v) prints the results when possible.

The interpretation of the request is achieved taking into account the vocabulary and the syntax of the service components included in the service catalogue of the user. Figure 34 shows the SPATEL logic generated for the interpretation of the sentence "My preferred news translated in english" (in French "Mes infos traduites en anglais").

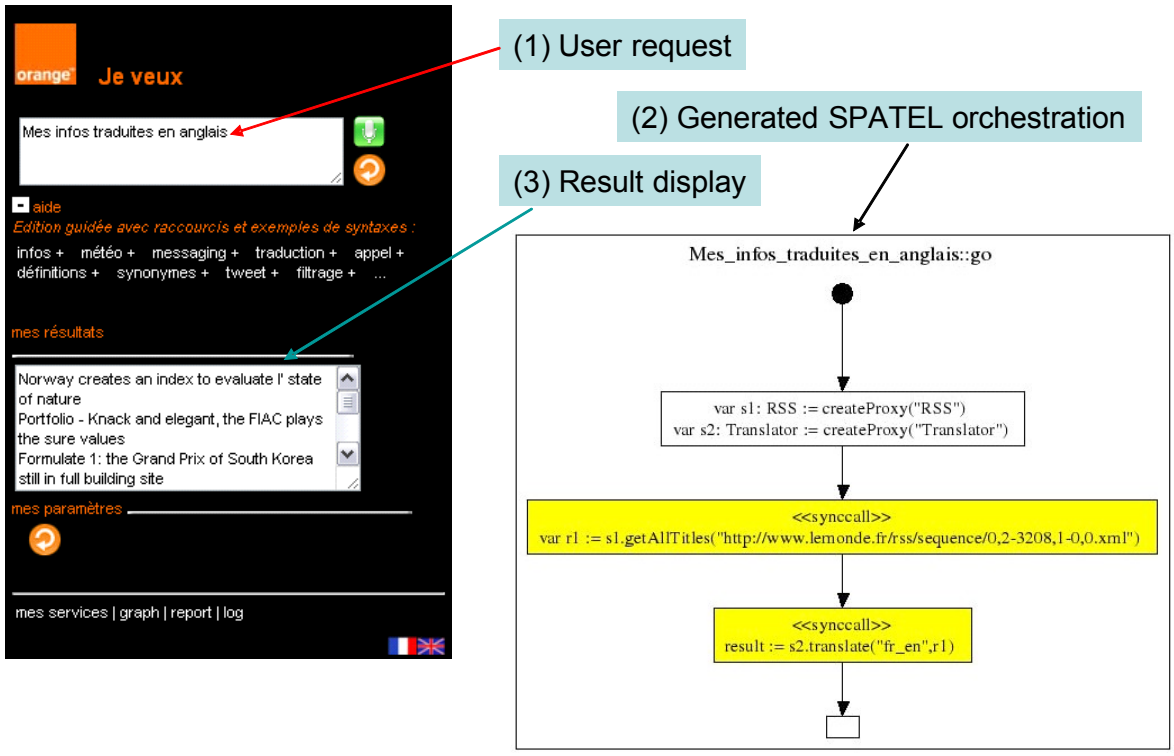


Figure 40: Automatic orchestration from natural language request

Natural language annotations (that's to say, the vocabulary and the syntax patterns to match a service included in the catalogue) are defined declaratively in configuration files but most of the rules require an explicit coding to take into account some specificities like disambiguation. The Figure below shows service configuration for 'translation' service.

```
RULES = {
    'en' : {
      'verbs': ('translate',),
      'nouns' : ('translation',),
      'modifiers' : ('translated',),
      'complement' : (),
      'args' :
        'sourceLanguage' : {
           'type' : 'Language',
           'by_prefix' : ("from",),
           'as_adjectif' : False,
          },
        'targetLanguage' : {
           'type' : 'Language',
           'by_prefix' : ("in","to"),
           'as_adjectif' : False,
       },
      },
```

Figure 41: Excerpt of natural language configuration for a service

A noticeable feature of the interpretation system is that all natural language annotations and the reasoning are done on *abstract* service definitions, not concrete service definitions. This is the basis for achieving context aware services. For instance SMS sending abstract service will be defined independently of the concrete service API offered by a telecom provider, such as Orange or Telefonica. In the target execution environment (SPATEL Engine) an implementation variant of the SMS service operation (see variability mechanism in Section 3.2.3.2) will perform user context resolution at runtime before deciding which concrete implementation variant to call.