# Table des matières

# Introduction

Les systèmes et environnements ambiants consistent en un ensemble de dispositifs mobiles ou fixes connectés ensemble à travers divers réseaux de communication. Ces dispositifs hébergent des composants logiciels développés, installés et activés de manière indépendante. Ces composants peuvent fournir des services, et, à leur tour, peuvent nécessiter d'autres services pour fonctionner. Un composant peut ainsi posséder un ou plusieurs services requis ou fournis (éventuellement les deux) qui constituent ses interfaces. Ces composants sont des "briques" logicielles qui peuvent être assemblées pour réaliser des applications possédant des services plus complexes qui sont offerts à un utilisateur.

En raison de la mobilité des composants et des utilisateurs, l'environnement ambiant est ouvert et instable : des dispositifs et des composants logiciels peuvent apparaître et disparaître sans que cette dynamique ne soit nécessairement prévue ou anticipée. Dans ce contexte, les besoins de l'utilisateur ne sont pas non plus stables et peuvent varier rapidement. Les utilisateurs se placent au centre de ces systèmes dynamiques et peuvent utiliser les services qui sont mis à leur disposition. L'intelligence ambiante vise à leur offrir un environnement personnalisé, adapté à la situation, à anticiper leurs besoins et à leur fournir "les bonnes applications au bon moment".

Notre équipe développe actuellement une solution dans laquelle les composants logiciels sont assemblés de manière dynamique et automatique afin de construire des applications composites et ainsi personnaliser l'environnement au moment de l'exécution. Cette approche est appelée "composition opportuniste" [Triboulot et al., 2015]. Contrairement au mode traditionnel "top-down" pour la construction d'applications, la composition opportuniste construit les applications à la volée, en mode "bottom-up", à partir des composants qui sont présents et disponibles au moment de l'exécution, c'est-à-dire en fonction des opportunités, et non sur la base de besoins explicites. Par exemple, un composant d'interaction

standard présent dans un smartphone (*p. ex.*, un curseur glissant ou un composant de reconnaissance vocale), un convertisseur logiciel et une lampe connectée peuvent être opportunément assemblés et fournir à l'utilisateur un service de contrôle de la lumière ambiante lorsqu'il entre dans une pièce. De telles applications peuvent être imprévues et inconnues de l'utilisateur, d'où la nécessité de l'en informer ; il peut alors décider si l'application répond à un besoin actuel.

Placé au centre de ces systèmes ambiants, l'utilisateur doit garder le contrôle sur son environnement. Cette tâche doit être prioritaire [Bach and Scapin, 2003] : les utilisateurs doivent pouvoir configurer leur environnement et être assistés autant que possible dans cette tâche. En l'absence de spécification préalable, les applications qui émergent de l'environnement sont inconnues *a priori*, inattendues, parfois même surprenantes. Pourtant, l'utilisateur doit en être toujours conscient de ces émergences. Ainsi, le système doit être capable de gérer les problèmes suivants : (i) présenter les applications d'une manière utile et compréhensible pour informer efficacement l'utilisateur (ii) que l'utilisateur dispose de fonctions et de facilités lui permettant de contrôler et d'agir sur la configuration de ces applications.

La composition opportuniste est supportée par un moteur de composition (autrement appelé moteur d'assemblage) [Younes et al., 2018] conforme aux principes du calcul autonome et au modèle MAPE-K [Kephart and Chess, 2003] : le moteur détecte les composants existants, décide des connexions (il peut connecter un service requis et un service fourni si leurs interfaces sont compatibles) sans utiliser un plan pré-établi. Le cœur de ce moteur est un système multi-agent distribué où les agents, au plus proche des composants logiciels, coopèrent et décident des connexions entre leurs services. Contrairement au paradigme traditionnel de développement du logiciel, l'utilisateur ne spécifie pas un besoin que l'application à construire doit satisfaire ("mode pull"). Ici, des applications adaptées au contexte lui sont fournies en "mode push" après avoir été construites en l'absence de besoins explicitement définis par l'utilisateur. Pour cela, pour pouvoir prendre des décisions pertinentes et proposer les bonnes applications à l'utilisateur, le moteur de composition intelligent apprend automatiquement les préférences de l'utilisateur en fonction de la situation. L'apprentissage se fait par renforcement [Sutton and Barto, 2018] à partir des données de l'utilisateur qui est mis dans la boucle [Younes et al., 2020]. Par conséquent, il faut être capable d'extraire et de traiter ces informations de feedback provenant de l'utilisateur. Ainsi, le moteur de composition assure la proactivité et l'adaptation de l'exécution dans un contexte ouvert,

dynamique et imprévisible.

Pour répondre à ces problèmes, nous proposons dans cette thèse une approche qui repose sur l'ingénierie dirigée par les modèles (IDM). L'idée initiale était de pouvoir transformer rapidement les résultats de composition fournis par le moteur de composition en représentations graphiques des applications composites. L'avantage de cette approche est qu'une fois le modèle de l'application obtenu, il est possible de : (i) fournir de multiples représentations structurelles de l'application composite ; (ii) générer des représentations complémentaires (*p. ex.*, une vue dynamique de l'application ou une description à base de règles), (iii) enregistrer formellement les manipulations du modèle par l'utilisateur ; et (iv) extraire le feedback de l'utilisateur nécessaire à alimenter l'apprentissage du moteur de composition. Notre solution consiste en un métamodèle et en un ensemble de langages et de processus de transformation de modèle. Nous proposons un éditeur qui permet à l'utilisateur d'être informé des applications émergentes, de comprendre leur fonction et leur utilisation, et de les modifier s'il le souhaite. À partir de ces actions, et sans surcharger l'utilisateur, l'éditeur est capable d'extraire son feedback et de le renvoyer au moteur.

L'originalité de notre approche réside dans la manière d'utiliser l'IDM et sa combinaison avec la composition opportuniste. L'IDM a principalement été créée pour assister les concepteurs et les développeurs d'applications. Grâce aux outils existants, ils peuvent modéliser et créer des applications, produire du code exécutable, sur la base du besoin de l'utilisateur. Dans notre approche, les outils de l'IDM sont destinés aux utilisateurs finaux (qui, la plupart du temps, sont des utilisateurs non experts) pour leur décrire les applications émergentes et leur permettre de manipuler eux-mêmes les modèles des applications.

## Publications

Ce travail a fait l'objet de différentes communications :

— Un poster, "Emergence of composite services in smart environments", primé à "Euro Science Open Forum" (ESOF 2018) classé parmi les 10 premiers parmi plus de 200 [Koussaifi et al., 2018]

— Un papier, "Ambient intelligence users in the loop : Towards a model-driven approach", publié dans le workshop international "Microservices : Science and Engineering" (MSE) associé à la conférence "Software Technologies : Applications and Foundations" (STAF) en 2018 [Koussaïfi et al., 2018]

— Un papier, "Automated user-oriented description of emerging composite ambient applications", publié dans la 31ème conférence internationale SEKE "Software Engineering and Knowledge Engineering" en 2019 [Koussaifi et al., 2019]

— Un papier, "User-oriented description of emerging services in ambient systems", publié dans le Ph.D. symposium de la 17ème conférence internationale ICSOC "Service-Oriented Computing" en 2019 [Koussaifi, 2019]

— Un rapport de recherche, "Putting the End-User in the Loop in Smart Ambient Systems : an Approach based on Model-Driven Engineering" produit en mai 2020 [Koussaifi et al., 2020]. Cette communication a été soumise à la revue "Journal of Universal Computer Science (J.UCS)" dans le cadre de l'appel à communication sur la thématique "Advances and Challenges for Model and Data Engineering" (en cours de révision).

## Organisation du manuscrit

Ce mémoire est organisé comme suit :

**Partie I : Contexte et problématique**

Cette première partie décrit le contexte de ce travail et la problématique, et analyse l'état de l'art.

— **Chapitre 1 : Contexte**

Les différents éléments du contexte sont introduits : le développement basé sur les composants, la composition opportuniste et le système de composition automatique, l'IDM et les différents outils utilisés pour mettre en œuvre notre solution.

— **Chapitre 2 : Problématique et exigences**

Notre problématique est exposée accompagnée des différentes questions de recherche. Les problèmes sont formalisés et présentés sous forme d'exigences que notre proposition doit satisfaire.

— **Chapitre 3 : État de l'art**

Différents travaux en lien avec les questions de recherche et les exigences formulées sont présentés. Le chapitre se termine par une analyse de ces travaux au regard des exigences que nous avons formulées.

**Partie II : Description automatisée centrée utilisateur des applications composites émergentes**

Cette partie présente en détail notre proposition.

— **Chapitre 4 : Approche basée sur l'IDM pour assister l'utilisateur dans son environnement**

Une vue d'ensemble de la solution proposée dans cette thèse est présentée. Plus précisément, les différentes contributions qui répondent à la problématique sont introduites. Le chapitre se termine par les motivations qui nous ont amené à choisir et à utiliser l'IDM.

— **Chapitre 5 : Présentation structurelle de l'application émergente**

Les différents éléments de la modélisation et de la présentation des modèles d'applications sont présentés : le métamodèle d'assemblage, le langage spécifique pour

visualiser la représentation structurelle de l'application et la transformation de modèle pour la présentation. Le chapitre se termine par une analyse de la proposition et de ses limites.

— **Chapitre 6 : Description multivue de l'application émergente**

Ce chapitre présente les contributions qui visent à présenter une application d'une manière plus riche, avec davantage de sémantique : un deuxième DSL graphique, les transformations de modèle pour obtenir plusieurs vues de l'application, et le métamodèle complet pour la description à base de règles.

— **Chapitre 7 : Rôle de l'utilisateur et feedback**

Le rôle de l'utilisateur et de ses interventions sont présentés ici. En outre, ce chapitre explique comment, grâce à l'IDM, les actions de l'utilisateur et son assistance dans la modification de l'application sont traitées. De plus, le chapitre présente l'extraction du feedback de l'utilisateur d'après ses interventions.

**Partie III : Développement, expérimentation et analyse**

Cette troisième partie décrit l'implémentation de la solution présentée dans la partie précédente, et présente les expériences réalisées pour valider notre proposition.

— **Chapitre 8 : Développement d'ICE**

Ce chapitre présente les différents développements effectués pour réaliser notre Environnement de Contrôle Interactif (ICE), et montre comment les différents outils de l'IDM sont utilisés.

— **Chapitre 9 : Intégration avec OCE**

La solution d'intégration d'ICE avec le moteur de composition opportuniste (OCE) est présentée. En outre, le chapitre présente les différentes étapes du développement de cette solution. Le chapitre se termine par une analyse des perspectives d'amélioration de la solution.

— **Chapitre 10 : Expérimentation**

Ce chapitre montre les expériences réalisées pour valider les contributions de cette thèse. Les expérimentations couvrent plusieurs formes d'architecture des applications émergentes.

— **Chapitre 11 : Analyse de la contribution**

Ce chapitre analyse les résultats des expérimentations et fait un point d'avancement sur nos réponses aux exigences. En outre, le chapitre présente l'originalité de notre approche et l'apport de l'utilisation de l'IDM dans notre contexte.

**Conclusion et perspectives**

La conclusion globale de cette thèse synthétise les problèmes et les solutions proposées pour les résoudre. En complément, un certain nombre de problème ouverts et de perspectives d'amélioration de notre solution sont discutés.

*Première partie*

# Contexte et problématique

# 1 Contexte

Ce chapitre présente les différents éléments du contexte de la thèse. Tout d'abord, la notion de développement basé sur l'utilisation de composants logiciels est présentée. Les principes et objectifs de la composition logicielle opportuniste sont ensuite introduits, avec une vue d'ensemble de l'architecture de notre système de composition. Puis, une introduction sur l'ingénierie dirigée par les modèles et ses différents concepts est donnée. Enfin, les différents outils utilisés pour mettre en oeuvre notre solution sont présentés.

## 1.1 Ingénierie logicielle à base de composants

En 1968, Douglas McIllroy a introduit la notion de "composant logiciel" [1] pour la première fois lors de la première conférence internationale sur le génie logiciel (NATO International Conference on Software Engineering) [McIlroy et al., 1968]. Il en existe aujourd'hui plusieurs définitions. C. Szyperski [Szyperski et al., 2002] a focalisé la définition du composant sur ses caractères principaux :

**Definition 1.** *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*

I. Sommerville [Sommerville, 2016] quand à lui a défini un composant comme suit :

**Definition 2.** *A software component is as a standalone service provider... This emphasises two critical characteristics of a reusable component :*

1. *The component is an independent executable entity. Source code is not available, so the component does not have to be compiled before it is used with other system components.*
2. *The services offered by a component are made available through an interface, and all interactions are through that interface. The component interface is expressed in terms of parameterised operations and its internal state is never exposed.*

1. Dans ce mémoire, pour simplifier, le terme "composant" sera fréquemment utilisé à la place de "composant logiciel".

En se basant sur ces définitions, nous pouvons retenir qu'un composant est une boîte noire représentant une entité logicielle indépendante, réutilisable, standardisée (c'est-à-dire développée conformément à un modèle de composant standard) et qui peut être composée avec d'autres composants pour créer une application. Il peut offrir certaines fonctionnalités (éventuellement aucune) et possède des dépendances (éventuellement aucune) à d'autres composants. En outre, un composant peut être un composant métier ou un composant d'interaction directe avec l'utilisateur.

Chaque composant possède au moins une interface qui définit une certaine fonctionnalité ou dépendance (appelée aussi "service"). Elle peut être soit fournie ou soit requise :

— **Service fourni :** définit une fonctionnalité du composant qui peut être invoquée par d'autres composants. Selon le formalisme **UML 2.0**, il est représenté graphiquement par un cercle relié par une ligne à l'icône du composant.

— **Service requis :** définit un service nécessaire pour le fonctionnement du composant et qui doit être fourni par un autre service provenant d'un autre composant. Selon le formalisme **UML 2.0**, il est représenté par un demi-cercle relié par une ligne à l'icône du composant.

La figure 1.1 montre la représentation graphique (UML 2.0) d'un composant métier "Desk" permettant la réservation d'une salle. Il possède trois services : deux requis ("Book" et "Notify") et un fourni ("Order"), les deux premiers étant nécessaires à la réalisation du troisième.

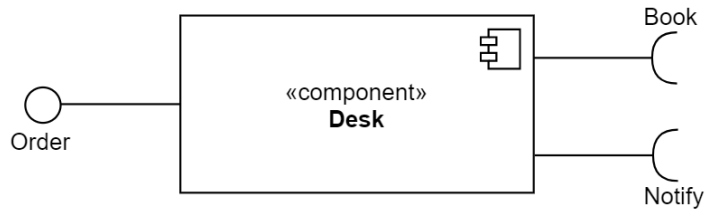


Figure 1.1 – Représentation en **UML 2.0** du composant **Desk** et de ses services

La figure 1.2 montre la représentation graphique d'un composant d'interaction "Slider" avec un seul service requis ("putValue").

Pour développer un composant, les ingénieurs peuvent utiliser une approche basée sur la programmation orientée objet (POO) [Cox, 1986]. La notion de service fourni par le composant se rapproche de la notion de méthode définie dans l'objet et qui peut être appelée

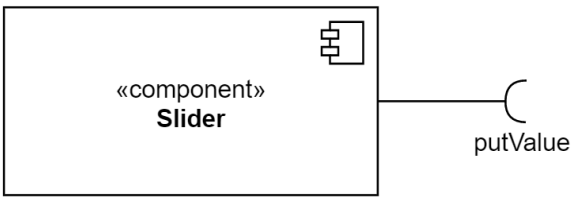


Figure 1.2 – Représentation en **UML 2.0** du composant **Slider** avec un service requis putValue

par d'autres objets. Cependant, les composants logiciels diffèrent des objets sur plusieurs aspects :

— Le découplage entre les codes (classes) qui implantent le composant appelant (demandeur du service) et le composant appelé (fournisseur du service) : un composant expose ses interfaces/services requis, mais pas l'objet.

— Dans le composant, un service requis est demandé sans expliciter qui le rend, c'est-à-dire sans désigner la référence (la communication est anonymisée).

— Le composant est donc une brique qui peut être composée avec d'autres, plus facilement qu'avec les objets, indépendamment du contexte dans lequel elle a été développée (réutilisation). Cela facilite la flexibilité du logiciel.

La programmation orientée composant (POC), en anglais *Component-based Software Engineering (CBSE)*, est apparue pour la première fois il y a une vingtaine d'années. Son objectif principal était de favoriser le développement de systèmes logiciels en se basant sur le principe de réutilisation. Par conséquent, les composants sont utilisés comme éléments de base : ils sont **composés** (assemblés) en connectant les services fournis avec les services requis adéquats (c'est-à-dire possédant le même profil) pour réaliser un **assemblage** qui réalise l'application. Ce processus est connu sous le nom de **composition** ou d'**assemblage** de composants.

Généralement, dans les approches CBSE, la construction des applications se fait à travers une approche descendante ("top-down") à partir d'un ensemble d'exigences qui sont exprimées par les parties prenantes, en particulier les besoins de l'utilisateur.

Ce processus est divisé en plusieurs étapes. Tout d'abord, après avoir défini les besoins (utilisateur, système, ressources, ...) et les exigences du logiciel, le développeur le divise en unités plus simples. Pour implémenter ces unités, le développeur cherche dans un premier temps s'il existe déjà des composants qui peuvent être re-utilisés pour programmer ces

dernières. Cependant, si aucun composant disponible ne peut répondre aux exigences, le développeur développe alors, dans un deuxième temps, des composants convenables. Finalement, après avoir sélectionné ou créé les composants nécessaires le développeur construit l'application en assemblant les composants.

Dans un assemblage de composants, un composant peut être remplacé par un autre à tout moment, éventuellement même à l'exécution. Par exemple, si un composant n'est plus disponible pour une raison quelconque, il peut être remplacé par le développeur au moment de la conception de l'application par un autre composant équivalent. De même, si au moment de l'exécution de l'application, un composant a disparu, il peut être remplacé par un composant candidat pour maintenir le service offert par l'application. Cette équivalence est établie sur la base de la compatibilité des services des deux composants : le service fourni par un composant doit être compatible avec le service requis de l'autre composant.

## 1.2 Composition opportuniste

Dans les approches traditionnelles, la composition de composants logiciels est réalisée par un ingénieur afin de construire une application qui répond à un besoin explicite préalablement défini. Pour répondre à ce besoin, des composants sont sélectionnés parmi des composants existants réutilisables ou développés spécifiquement, puis assemblés afin de former une application. Ce mode de développement est qualifié d'approche "top-down".

Cependant dans les environnements que nous considérons, qui sont dynamiques, ouverts et changeants, une approche "top-down" est difficile à adopter. En effet, plusieurs contraintes se posent dans ce type d'environnement :

1. Au niveau de l'expression du besoin : le besoin est changeant en fonction de la situation courante de l'utilisateur. De plus, il peut être difficile à exprimer pour ce dernier.

2. Au niveau du choix des composants à assembler, compte-tenu du nombre de composants qui peuvent peupler l'environnement et de leur volatilité (disponibilité) dans cet environnement dynamique et ouvert (des composants apparaissent, possiblement inconnus, et disparaissent en continu).

### 1.2.1 Principes et objectifs

La composition opportuniste [Triboulot et al., 2014] est une approche originale de développement logiciel, explorée dans notre équipe de recherche, qui a pour objectif de construire des applications à la volée par assemblage de composants logiciels qui sont disponibles sur l'instant dans l'environnement. Le Larousse définit l'opportunisme comme l'"attitude consistant à régler sa conduite selon les circonstances du moment, que l'on cherche à utiliser toujours au mieux de ses intérêts". Pour le CNRLT c'est la "ligne de conduite politique dans laquelle la tactique se détermine d'après les circonstances" [CNRTL, 2012]. Ces définitions mettent en évidence les notions de circonstance, d'intérêt et d'adaptation, que l'on retrouve dans notre approche. En effet, on peut définir les "circonstances" comme étant l'ensemble de composants logiciels (et leurs services) disponibles dans l'environnement à un moment donné et la composition opportuniste comme la capacité d'assembler ces composants en une application parce qu'ils sont en situation d'être composés et non, comme dans les approches traditionnelles, pour satisfaire un besoin exprimé *a priori* [Triboulot et al., 2015]. Dans cette approche dirigée par le contexte, les fonctionnalités émergent de l'environnement et le même moyen permet, en fonction des opportunités, de composer et de recomposer des applications qui sont donc adaptées aux circonstances.

Ici, l' "intérêt" de l'utilisateur n'est pas explicite : il est observé *a posteriori* afin de déterminer l'utilité de la composition, de décider de sa préservation et d'aider à la sélection de futures compositions pertinentes, c'est-à-dire adaptées à l'utilisateur et à la situation.

L'"adaptation" est permanente. Les nouveaux composants peuvent être intégrés dans des applications, sans qu'il soit nécessaire de modifier ou notifier quelque chose (comme refaire une décomposition ou exprimer un nouveau besoin). Les applications qui émergent sont ainsi adaptées aux circonstances.

Pour construire “les bonnes applications au bon moment”, il est nécessaire de contrôler ce qui émerge et de sélectionner les applications pertinentes. La composition doit être automatisée, et réalisée par un système que nous appelons “moteur de composition opportuniste”.

Par conséquent, la composition opportuniste offre plusieurs avantages :

1. **Réactivité :** Le moteur de composition opportuniste propose des applications utiles en fonction de la situation actuelle de l'utilisateur. Il cherche à réaliser des compositions

pertinentes avec les composants disponibles sans s'appuyer sur les besoins explicites de ce dernier, mais sur ses préférences préalablement apprises au fur et à mesure des compositions dans différentes situations.

2. **Flexibilité :** Les applications sont adaptées en continu à la situation actuelle, en remplaçant des composants parce qu'ils ont disparus ou parce que d'autres sont plus pertinents. Le système supporte l'ouverture et prend en compte l'apparition et la disparition des composants de l'environnement. D'autre part, les applications sont résilientes, car elles sont continuellement maintenues et améliorées.
3. **Génericité :** En plus de ne pas se baser pas sur des besoins de l'utilisateur et les plans d'assemblage prédéfinis, l'approche opportuniste ne suppose aucune hypothèse sur la présence des composants, ni sur leurs types ni sur leurs logiques métiers.

Cependant la composition opportuniste demande de répondre à un certain nombre de questions parmi lesquelles : 1. l'identification des composants et la sensibilité au contexte 2. la sélection des composants et l'automatisation de la composition 3. la prise en compte de l'intérêt de l'utilisateur.

### 1.2.2 Cas d'utilisation

MK travail dans un laboratoire de recherche occupant un poste de chef de projet. Il souhaite organiser une réunion avec ses collaborateurs et doit réserver une salle. Le laboratoire a préalablement déployé les composants suivants dans l'environnement ambiant :

— un composant de réservation (**Desk**) offrant le service *Order* et requérant deux services *Book* et *Notify*.

— un planificateur de salle (**Planner**) offrant le service *Book*.

Sur l'ordinateur de MK, les composants suivant sont installés :

— un dispositif d'entrée vocal (**Voice**) requérant le service *Order*.

— un dispositif d'entrée textuel (**Text**) requérant le service *Order*.

— un gestionnaire de calendrier (**Calendar**) qui offre le service *Notify*.

Nous présentons dans ce qui suit deux scénarios pour ce cas d'utilisation.

**Scénario 1.** MK allume son ordinateur et trouve dans son environnement les composants d'interaction (Voice et Text) et les composants métiers (Desk, Planner et Calendar) comme illustré dans la figure 1.3.

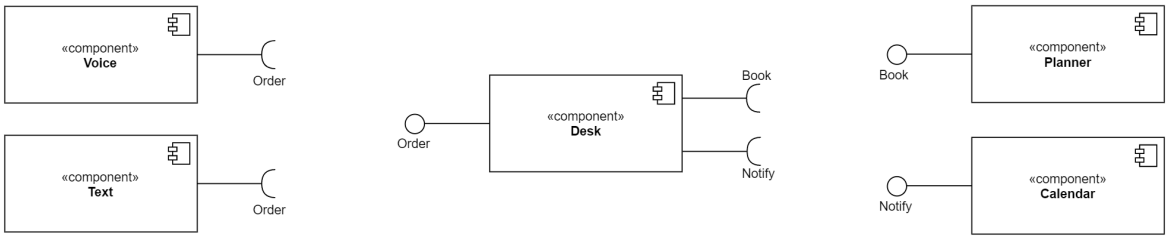


Figure 1.3 – Composants et services présents dans l'environnement

En utilisant l'approche basée sur la composition opportuniste, des composants convenables sont détectés et sélectionnés parmi ceux qui sont disponibles dans l'environnement ambiant, puis, ils sont composés pour faire émerger une application de réservation de salle de réunion et la proposer à MK. La structure de cette dernière est représentée par le diagramme de composants de la figure 1.4. Les connexions entre les services sont représentées avec une flèche rouge.

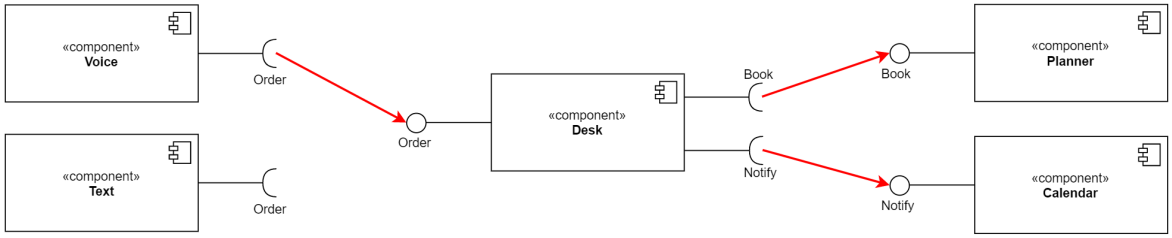


Figure 1.4 – Assemblage d'une application de réservation de salle de réunion

**Scénario 2.** À un certain moment durant l'exécution de l'application de réservation, le composant **Voice** tombe en panne et ainsi disparaît de l'environnement ambiant. Cependant, l'application de réservation est recomposée automatiquement puisque le système de composition opportuniste a détecté que le composant **Text**, qui peut remplacer le composant **Voice**, est toujours présent dans l'environnement. La nouvelle application est proposée à MK qui peut continuer à l'utiliser. Le nouvel assemblage qui réalise cette application est représenté dans la figure 1.5

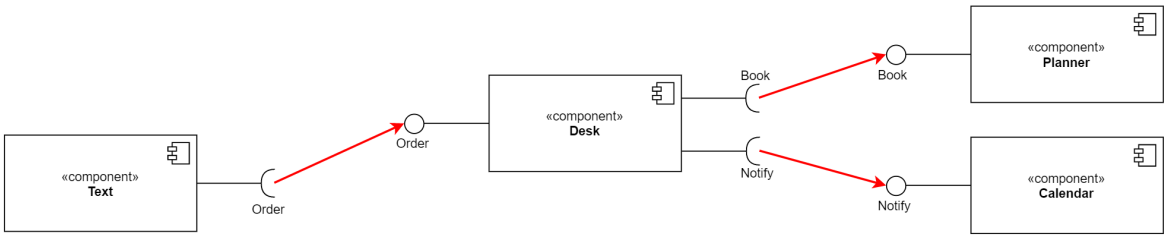


Figure 1.5 – Assemblage d'une application de réservation de salle de réunion (après disparition du composant **Voice**)

Ainsi, en utilisant la composition opportuniste, le système a réussi à s'adapter à la si-

tuation de l'environnement et, à la disparition du composant, à continuer à offrir un service pertinent.

### 1.2.3 Vue d'ensemble du système de composition

Le moteur de composition opportuniste (OCE) [Younes et al., 2018] est une unité indispensable à la solution. Il construit les applications émergentes qui seront présentées à l'utilisateur.

Le coeur d'OCE est un système multi-agent (SMA) distribué [Wooldridge, 2009] dans lequel chaque service d'un composant est contrôlé et administré par un agent. De manière générale, les agents sont des entités autonomes qui coopèrent afin d'atteindre un objectif commun. Ici, cette coopération mène à établir des connexions entre les différents services disponibles pour créer des assemblages. Les applications (réalisées par les assemblages de composants) émergent continuellement de l'environnement et des interactions entre agents, en profitant des opportunités au fur et à mesure qu'elles se présentent.

Pour prendre les bonnes décisions et construire des applications pertinentes et appropriées, OCE (plus précisément les agents qui le constituent) doit disposer d'informations utiles dans un contexte où les besoins n'ont pas été explicités. Ainsi, OCE apprend les besoins de l'utilisateur [Younes et al., 2020] durant l'exécution du système par un système d'apprentissage par renforcement [Sutton and Barto, 2011] alimenté par un "feedback" de l'utilisateur, qui représente les interventions faites par l'utilisateur sur l'application proposée.

La figure 1.6 présente un vue d'ensemble sur l'architecture globale de notre système.

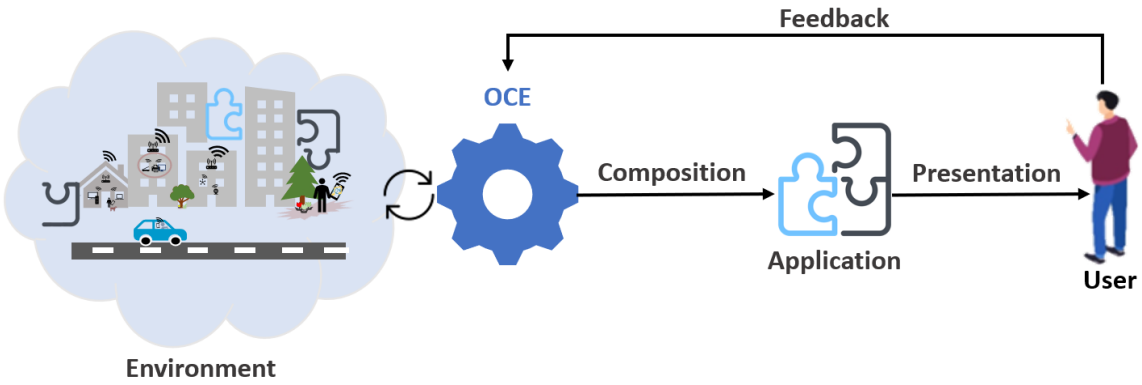


Figure 1.6 – Vue d'ensemble de l'architecture du système

OCE, tout d'abord, détecte les composants disponibles dans l'environnement. Puis,

après avoir sélectionné les meilleurs candidats, il compose une application qui est présentée à l'utilisateur et qui lui offre un service qui peut répondre à son besoin. L'utilisateur peut ensuite faire ses interventions et prendre une décision à utiliser l'application ou à la rejeter. D'après ces interventions, le système extrait le feedback et l'envoie à OCE pour mettre à jour sa connaissance. Ceci va lui permettre d'apprendre le besoin de l'utilisateur et de lui proposer des applications plus pertinentes.

**Dans cette thèse, le focus est sur l'utilisateur, sa place et son rôle dans le système et l'environnement.** En parallèle, un autre travail de thèse dans notre équipe se concentre sur OCE, la composition et le SMA.

## 1.3 Développement et programmation par l'utilisateur final

La programmation par l'utilisateur final, en anglais « End-User Programming » (EUP), propose un ensemble de techniques et d'outils qui permettent aux utilisateurs finaux de créer ou de modifier des applications pour répondre à leurs besoins [Barricelli et al., 2019]. Les utilisateurs finaux sont généralement des personnes qui n'ont pas de compétences avancées dans le domaine du développement logiciel. L'EUP entre dans le cadre plus large du développement par l'utilisateur final, en anglais « End-User Development » (EUD), qui vise à permettre aux utilisateurs finaux de participer à la conception et au développement des applications. Cette implication des utilisateurs est effective non seulement au moment du développement mais aussi au moment de l'exploitation de l'application. Selon F. Paternò [Paternò, 2013], qui critique plusieurs travaux concernant notamment les applications mobiles, la motivation est que les cycles de développement réguliers sont trop lents pour répondre aux besoins des utilisateurs qui évoluent rapidement.

Selon leurs préférences, les utilisateurs peuvent choisir d'utiliser le langage naturel ou des techniques de programmation visuelles ou simplifiées tels que Scratch ou Blockly pour développer leurs solutions [Bau et al., 2017]. Les approches de type "no-code" [Bubble, 2012, Google, 2014] aident les utilisateurs à développer leurs applications à travers des interfaces graphiques. En outre, il existe des solutions qui proposent un ensemble d'outils qui permettent aux utilisateurs finaux sans expérience en programmation de personnaliser le comportement de leurs applications en fonction du contexte grâce à la spécification de règles de déclenchement d'actions [Ghiani et al., 2017].

Par ailleurs, il existe des approches qui se basent sur l'IDM pour supporter l'EUP/EUD [Bucchiarone et al., 2019]. Les techniques et outils offerts par l'IDM augmentent le taux de participation des utilisateurs dans la conception et le développement des applications.

## 1.4 Ingénierie dirigée par les modèles

Cette section présente d'une façon générale l'Ingénierie Dirigée par les Modèles (IDM) et son approche pour résoudre un problème. Elle présente les différents concepts de l'IDM et l'outillage utilisé dans cette thèse.

### 1.4.1 Introduction

L'Ingénierie Dirigée par les Modèles est une méthodologie de développement de logiciels qui consiste en un ensemble de pratiques qui se concentrent sur la création et l'exploitation de modèles liés à un problème spécifique [Schmidt, 2006]. L'objectif est de permettre aux Développeurs de Logiciels (DL) de créer des descriptions abstraites d'un logiciel et de faciliter la génération du code de mise en œuvre [Combemale et al., 2016]. Ainsi, les modèles exploités par l'IDM ne sont pas utilisés seulement pour documenter le logiciel en question, mais ils deviennent les principaux artefacts du développement [Bézivin, 2006, Schmidt, 2006]. Ces derniers sont affinés afin d'être transformés en produit final.

Pour résoudre un problème en utilisant l'IDM, les développeurs suivent une approche qui combine la définition et l'utilisation de plusieurs éléments et de technologies. Cette approche, connue sous le nom de l'Architecture Dirigée par les Modèles, en anglais "Model Driven Architecture" (MDA), rend la phase de développement plus souple grâce à l'utilisation importante des modèles. Ces derniers peuvent être exprimés au moyen de plusieurs formalismes, comme UML [OMG, 2020b]. De plus, une approche IDM se base aussi sur la notion de transformation de modèle qui permettent de passer d'un espace technique à un autre, par exemple, transformer un modèle UML en un document XML. La figure 1.7 représente une vue d'ensemble sur l'architecture MDA.

Les phases fondamentales de cette approche sont :

1. Définir un métamodèle qui formalise le problème à résoudre. Cette phase est appelée métamodélisation.

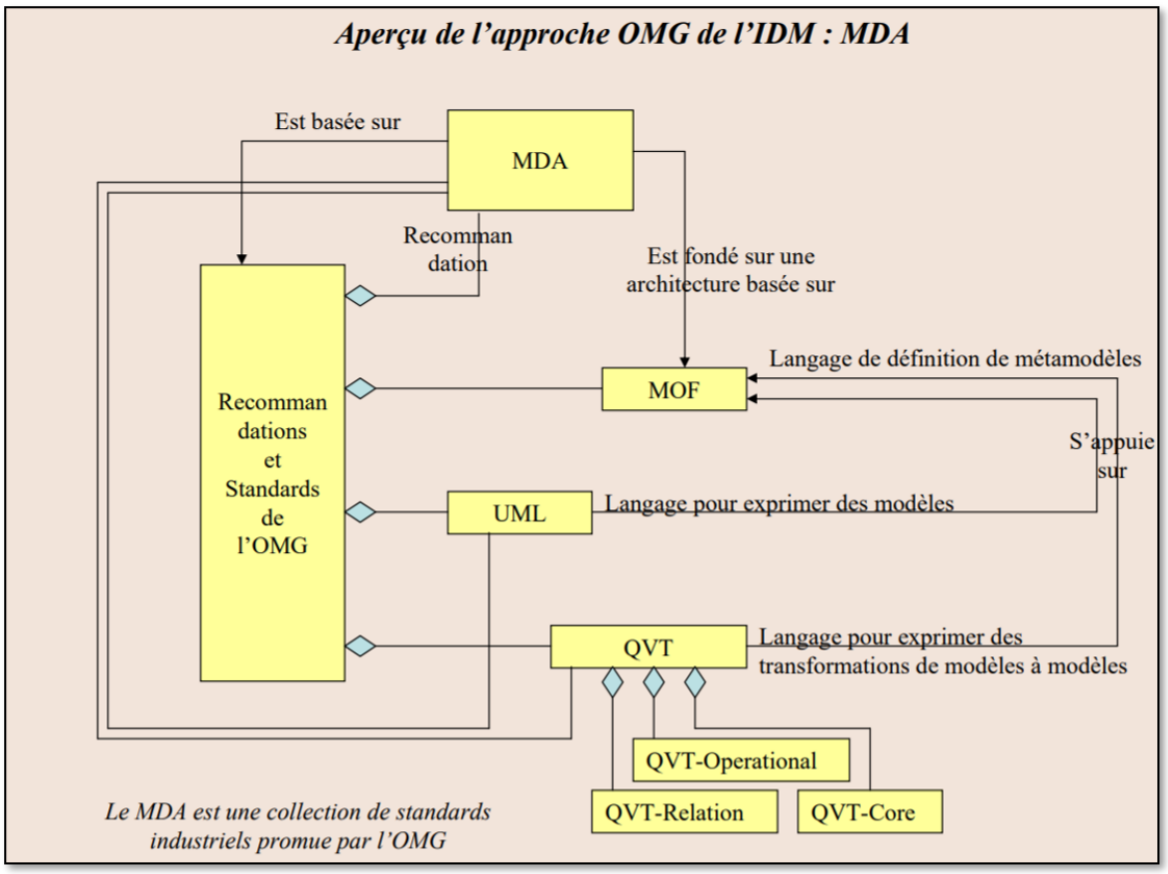


Figure 1.7 – Architecture Dirigée par les Modèles (Source : [Rousse, 2007])

2. Définir un ou plusieurs langages pour exprimer les modèles sous forme d'une représentation souhaitée. Ces derniers sont utilisés pour exprimer et résoudre un problème spécifique à un domaine. Ces langages sont appelés langages spécifiques à un domaine (DSL).

3. Définir un ou plusieurs outils de transformation de modèle pour transformer le modèle de la solution en d'autres modèles (pour le déploiement par exemple).

Ces éléments sont définis en détail dans les sections suivantes.

#### 1.4.1.1 Métamodélisation

Au coeur de l'IDM, les langages de modélisation, UML par exemple, sont utilisés pour définir la syntaxe abstraite et la syntaxe concrète du modèle [Vogel, 2018]. Ces derniers sont utilisés par les développeurs pour formaliser la structure de l'application, son comportement et ses exigences dans un domaine particulier. Cette étape vise à définir un métamodèle qui sera utilisé pour créer des modèles qui correspondent à l'architecture de ce dernier.

La syntaxe abstraite consiste à définir les différents éléments du modèle de l'application en précisant les relations entre eux. Ces éléments, combinés ensemble, créent une instance de l'application.

La syntaxe concrète d'un langage est la notation lisible et utilisée par le DL pour créer et représenter un modèle. Elle peut être graphique ou textuelle. Cette syntaxe est utilisée pour définir et personnaliser les attributs nécessaires pour les éléments modélisés, ce qui permet de définir comment le logiciel apparaîtra après sa génération.

Par ailleurs, pour vérifier si le modèle est valide et correspond au métamodèle, des règles sont associées au métamodèle. Cette action permet de détecter et prévenir de nombreuses erreurs avant la phase de génération de code. Pour définir ces règles, les DL utilisent des langages à base de contraintes, comme OCL.

#### 1.4.1.2 Langage spécifique à un domaine

Un langage spécifique à un domaine (DSL) est un langage de programmation, de modélisation ou de spécification qui permet d'exprimer un problème dans un domaine spécifique [Baciková et al., 2013]. Ces langages étant destinés à être utilisés par les humains [Fowler, 2010], ils doivent être compréhensibles pour tous leurs utilisateurs, mais aussi être exécutables par la machine.

Un DSL est utilisé pour définir les actions qui permettent à un utilisateur de manipuler et d'éditer le modèle représenté. Ces actions peuvent être contrôlées à la volée grâce à la définition des conditions d'édition incluses dans le DSL. Ces dernières permettent d'éviter des erreurs de validation.

#### 1.4.1.3 Transformation de modèle

Dans le contexte de l'IDM, la transformation de modèle joue un rôle fondamental pour la manipulation de modèles. Elle permet de générer à partir d'un seul modèle plusieurs types d'artefact [Combemale et al., 2016] tels que le code source de l'application modélisée, les descripteurs de déploiement ou des représentations alternatives du même modèle de l'application. Dans l'IDM, il existe de multiples familles de langages qui définissent les règles et démarches à suivre pour transformer un modèle. Par exemple la famille QVT [OMG, 2015] contient des langages de transformation de modèles à base de requêtes.

Dans ce contexte il existe deux types de transformation de modèle [Vogel, 2018, Sendall and Kozaczynski, 2003]. Le premier correspond à la transformation endogène dont le résultat est un modèle qui appartient au même domaine de modélisation que le modèle transformé, c'est-à-dire que les deux modèles sont conformes à un même métamodèle. Le deuxième correspond à la transformation exogène dont le résultat est un modèle qui est conforme à un autre métamodèle que celui du modèle d'entrée.

#### 1.4.1.4 Éditeur

Un éditeur est un outil qui est construit par les DL en s'appuyant sur les concepts définis précédemment. C'est l'outil à travers lequel les modèles sont affichés et manipulés par les utilisateurs. Il peut être graphique ou textuel en fonction du DSL choisi.

### 1.4.2 Outillage

Dans cette section les différents outils et technologies de l'IDM utilisés dans le cadre de cette thèse sont présentés.

#### 1.4.2.1 GEMOC

GEMOC [Gemoc Initiative, 2012] est l'outil choisi pour développer les contributions de cette thèse. C'est un logiciel de développement d'applications (basées sur l'IDM) qui inclut plusieurs langages de modélisation. Grâce à GEMOC, les DL peuvent utiliser des techniques et des outils qui facilitent la création, l'intégration et le traitement automatisé de modèles.

Par comparaison avec les autres logiciels concurrents, comme Eclipse IDE [Eclipse, 2020c], les DL doivent explicitement chercher les extensions nécessaires, les télécharger et les installer pour qu'ils puissent développer un projet de modélisation. En revanche, GEMOC englobe, par défaut, toutes les extensions liées à la modélisation logicielle. Il est conçu de manière à faciliter la modélisation logicielle et la réalisation des projets de modélisation. Une autre raison pour avoir choisi GEMOC est que mon équipe a participé à sa création et contribue au programme "GEMOC initiative".

#### 1.4.2.2 Ecore

Ecore est un langage de modélisation appartenant à EMF [Eclipse, 2020d] qui permet de créer des métamodèles [Eclipse, 2020f] représentés sous forme de diagrammes de classe

UML. Il permet de définir les relations entre les éléments représentés et de définir les règles OCL nécessaires pour la phase de validation des modèles créés.

#### 1.4.2.3 Sirius

Sirius est un langage qui permet de créer facilement un DSL et un éditeur de modèles [Eclipse, 2020g]. Ceci permet d'initialiser et de représenter des instances du métamodèle Ecore développé.

Pour réaliser ces derniers, Sirius dispose d'un éditeur graphique, simple à utiliser, qui permet de définir les différents éléments du DSL et de créer les éditeurs personnalisés.

#### 1.4.2.4 Acceleo

Acceleo est un langage de transformation de modèle [Eclipse, 2020a] qui appartient à la famille Requête/Vue/Transformation, en anglais "Query/View/Transformation" (QVT). Il est utilisé comme générateur de code source permettant de mettre en oeuvre l'architecture MDA.

Il permet de transformer un modèle en texte en suivant une transformation endogène ou exogène. Le type de transformation dépend du paramétrage du langage lors de sa définition dans le programme.

# 2 Problématique et exigences

Ce chapitre détaille et précise la problématique et le contexte de cette thèse, en particulier en ce qui concerne la position et le rôle de l'utilisateur dans le système ambiant. Les différents problèmes sont formalisés sous la forme d'exigences que notre solution doit satisfaire.

## 2.1 Problématique

Les applications de l'Internet des objets et des systèmes ambiants consistent en un ensemble de composants physiques et logiciels, fixes ou mobiles, connectés. Logiciellement, ils peuvent être implémentés par des "composants logiciels" qui sont développés et administrés de manière indépendante. Ces derniers peuvent être assemblés (composés) pour créer ces applications. En raison de la mobilité et de l'instabilité de l'environnement ambiant, les dispositifs physiques et les composants logiciels peuvent apparaître et disparaître dynamiquement sans que cette dynamique ne puisse être anticipée. Par conséquent, l'environnement est ouvert et ses changements ne sont pas contrôlés.

Les utilisateurs sont au centre de ces systèmes dynamiques et ouverts où ils peuvent utiliser les applications à leur disposition et profiter des services qui sont offerts. L'intelligence ambiante vise à leur offrir un environnement personnalisé adapté à la situation courante et à leurs besoins, c'est-à-dire à leur fournir "les bonnes applications au bon moment", ceci en leur demandant le moins d'efforts possible.

Dans le cadre d'un projet de recherche en cours, notre équipe développe une solution originale pour construire ces applications ambiantes : les composants logiciels présents dans l'environnement ambiant sont assemblés dynamiquement et automatiquement pour faire émerger des applications [Triboulot et al., 2014] et ainsi personnaliser l'environnement au moment de l'exécution. Ce processus d'émergence est déclenché par la disponibilité des composants dans l'environnement et ne répond pas directement à un besoin explicite ex-

primé à l'avance par l'utilisateur. Par conséquent, les applications émergentes peuvent être imprévues et inconnues de ce dernier.

La composition opportuniste est supportée par un système "intelligent" appelé "moteur de composition opportuniste" ("Opportunistic Composition Engine", en abrégé OCE) [Younes et al., 2019a, Younes et al., 2020], conforme aux principes de l'informatique autonomique ("autonomic computing") et à l'architecture MAPE-K [Kephart and Chess, 2003] : OCE détecte les composants présents dans l'environnement, décide et planifie des assemblages de composants en fonction des opportunités, et enfin les propose à l'utilisateur. Cependant, OCE ne fait pas émerger toutes les applications possibles. Son objectif est de faire émerger celles qui sont pertinentes pour l'utilisateur. Pour cela, l'intelligence d'OCE repose sur l'apprentissage du besoin de l'utilisateur en fonction de la situation.

Le cas d'utilisation, présenté ci-dessous, permet d'illustrer notre problématique. Il est repris tout au long du mémoire por servir d'exemple : l'habitant d'une maison intelligente rentre chez lui. À son arrivée, le moteur OCE construit une application composée d'un composant logiciel d'interaction "Curseur" (ou"Slider") existant sur son smartphone, d'un convertisseur et d'une lampe connectée qui se trouve dans la pièce où il est actuellement. Le convertisseur a pour rôle de transformer le signal reçu du "Curseur" (par exemple une valeur entre 0 et 100) en un signal compatible avec la lampe (par exemple, "on" ou "off").

Une fois cette étape réalisée, l'utilisateur reçoit une notification sur son smartphone qui signale qu'un nouveau modèle d'assemblage est disponible dans son éditeur d'application. Ceci conduit aux questions de recherche suivantes :

**Question 1.** Comment un utilisateur au sein d'un système ambiant peut-il prendre connaissance des applications qui émergent et dont il n'a pas exprimé le besoin ?

**Question 2.** Comment l'utilisateur peut-il comprendre le service rendu par une application et comment peut-il l'utiliser ?

Une fois informé, l'utilisateur peut visualiser le modèle de l'application proposé, faire ses interventions afin de mettre en service l'application ou la rejeter. Cette tâche présente un autre problème exprimé par la question suivante :

**Question 3.** Quel rôle et quel contrôle donner à l'utilisateur et comment répartir les responsabilités entre lui et le moteur OCE ?

Les sections suivantes analysent les questions énoncées ci-dessus. Tout d'abord, la section 2.1.1 analyse le problème de la position et du rôle de l'utilisateur dans le système. Puis,

la section 2.1.2 analyse le problème de la présentation et de la description de l'application dédiée à l'utilisateur. Enfin, la section 2.1.3 analyse le problème de la capture du feedback de l'utilisateur.

### 2.1.1 L'utilisateur au centre de la boucle

Dans ce contexte d'intelligence artificielle, le partage des responsabilités décisionnelles entre OCE et l'utilisateur est en question. Selon C. Evers *et al.* et [Faure et al., ], dans les environnements ambiants, l'utilisateur doit se placer au centre où il est l'acteur principal qui contrôle les applications.

Avec notre approche, en l'absence de besoins préalablement explicités, il est nécessaire non seulement de l'informer des applications qui émergent dans son environnement, mais aussi de lui laisser le contrôle sur ces dernières. Ainsi, il doit pouvoir accepter ou rejeter les applications proposées par le moteur. Éventuellement, l'utilisateur doit pouvoir éditer et modifier les assemblages proposés pour construire des applications qui répondent à ses besoins. Il pourrait aussi avoir la possibilité de construire lui-même des applications sans l'aide du moteur. Enfin, après l'acceptation de l'application par l'utilisateur, cette dernière doit être déployée dans son environnement pour qu'il puisse bénéficier des services qui lui sont offerts.

La figure 2.1 détaille la relation entre l'environnement, le moteur et l'utilisateur. Elle synthétise les problèmes principaux de cette thèse (indiqués en mauve dans le schéma) : la présentation de l'application émergente à l'utilisateur - le positionnement de l'utilisateur, son rôle et ses interventions - le feedback de l'utilisateur.

Reprenons le cas d'utilisation présenté dans la section précédente. Après avoir construit une application composée d'un curseur, d'un convertisseur et d'une lampe (en réalité, le système construit un modèle de configuration et non pas l'application proprement dite), le système doit présenter cette dernière à l'utilisateur à travers un outil dédié. L'utilisateur, à son tour, doit pouvoir intervenir sur l'application avant de décider de l'accepter pour la déployer ou de la rejeter. Par exemple, si cette application lui convient, l'utilisateur doit pouvoir indiquer qu'il l'accepte pour être déployée dans son environnement. Cependant, il peut arriver que l'application proposée par le moteur ne convienne pas au besoin actuel de l'utilisateur. Ce dernier doit donc aussi posséder le privilège de la modifier pour l'ajuster à ses besoins. En outre, s'il ne souhaite pas ou ne veut pas la modifier, il peut la rejeter.

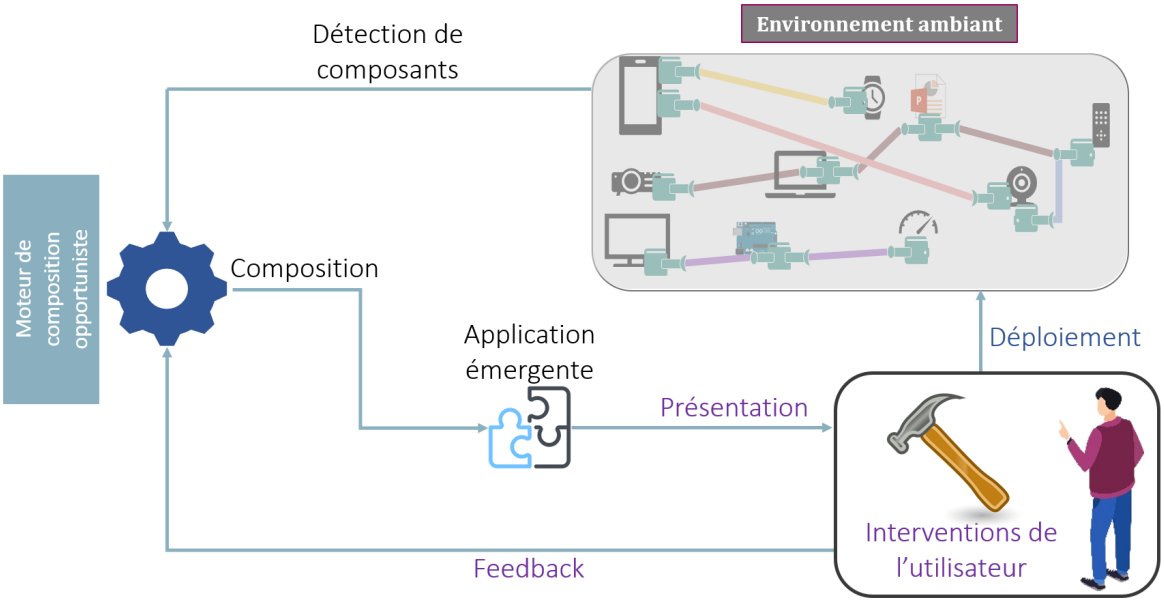


Figure 2.1 – Architecture présentant les différentes problématiques du système

Sachant que l'application proposée ne répond pas à une demande explicite de l'utilisateur, ce dernier n'attend pas forcément une proposition et peut ne pas comprendre de quoi il s'agit. Pour éviter cela, le système doit donc permettre de comprendre l'application présentée et le service qu'elle offre.

Enfin, le système doit être capable de capturer les réactions de l'utilisateur suite à une proposition dans le but d'en extraire des informations utiles à OCE pour ses décisions futures. De plus, cette tâche doit être réalisée sans surcharger l'utilisateur, c'est-à-dire que le système doit extraire ces informations d'une manière non intrusive. Comme la prise en compte des besoins de l'utilisateur se fait *a posteriori* et non *a priori*, OCE apprend pour les prochaines compositions.

En conclusion, les problèmes suivant peuvent être soulevés :

1. Le premier consiste en la manière de présenter une application à l'utilisateur pour lui permettre de connaître les différents composants utilisés et de lui présenter explicitement le service offert par l'application à travers une description fonctionnelle.

2. Le deuxième consiste à guider l'utilisateur durant son intervention pour l'aider éventuellement à modifier son application avant son acceptation.

3. Le troisième consiste à extraire des informations de feedback qui permettront au moteur OCE d'apprendre les besoins de l'utilisateur.

Les sections suivantes analysent ces trois problèmes.

### 2.1.2 Présentation et description de l'application émergente

En l'absence de spécification préalable, les applications émergentes sont *a priori* inconnues. Elles résultent du travail du moteur intelligent qui propose des compositions de composants, qui n'ont pas obligatoirement été développés pour être composés ensemble, parce que l'environnement courant le permet. La composition repose sur des préférences et des habitudes apprises des utilisateurs et sur la correspondance entre les interfaces des composants.

Au centre du système, l'utilisateur doit être informé de l'émergence d'une application, de ses fonctionnalités et de la manière de l'utiliser. Ceci va lui permettre de comprendre l'application et de pouvoir considérer si elle répond à ses besoins pour l'accepter ou la rejeter. Par conséquent, les applications émergentes doivent être décrites de manière intelligible à l'utilisateur, qui peut être un utilisateur non familier avec les langages de programmation en particulier, ou avec le domaine de l'informatique en général. L'utilisateur peut, par exemple,être l'habitant d'une maison intelligente ou un voyageur utilisant les transports publics d'une ville intelligente.

Pour assister l'utilisateur, la description de l'application doit couvrir deux notions :

1. L'application doit être présentée à l'utilisateur d'une façon compréhensible. Ainsi, sa description doit être adaptée à son profil et composée de différentes descriptions fonctionnelles et structurelles.
2. L'application doit être modifiable par l'utilisateur pour ainsi lui laisser le contrôle sur ce qu'il veut utiliser.

#### 2.1.2.1 Représentation au moyen de plusieurs vues

Comme mentionné dans la section précédente, l'utilisateur n'est pas toujours expert dans le domaine de l'informatique, donc il ne sera pas capable de comprendre comment l'application a été conçue si cette dernière est présentée d'une façon complexe. Par conséquent, l'application devrait être présentée sous la forme appropriée, en fonction du profil de l'utilisateur.

Voici les questions de recherche associées à ce problème :

1. Comment adapter la présentation d'une application à des utilisateurs aux profils et compétences différents ?
2. Comment présenter une application à un utilisateur d'une façon plus riche, détaillée et compréhensible ?

#### 2.1.2.2 Description fonctionnelle de l'application

Pour illustrer ce problème, considérons le cas d'utilisation présenté dans la section 2.1. Lorsque l'application d'éclairage va être présentée à l'habitant d'une façon adaptée à ses compétences, il va pouvoir comprendre sa structure en termes de composants et services utilisés pour réaliser l'assemblage. Cependant, il peut ne pas savoir, avec cette seule représentation structurelle, le service offert par cette application ni comment l'utiliser.

Par conséquent, le problème ici réside dans la construction d'une description sémantique compréhensible à partir des composants participants qui, en complément de la représentation structurelle, aide l'utilisateur à mieux connaître son application. Plus précisément, il s'agit de savoir comment utiliser une application et quel service elle offre. Par exemple, l'utilisateur doit obtenir une description comme "Si tu déplaces le curseur du *Slider*, la lampe sera en état On ou Off".

Un autre question se pose alors, elle est liée aux descriptions des composants et de leurs services, qui seront utilisées lors de la construction de la description de l'application. Autrement dit, comment passer de descriptions unitaires à une description globale.

Dans cette thèse, les questions de recherche liées à ces deux problèmes sont les suivantes :

1. Quel est le langage à utiliser pour décrire efficacement les composants et leurs services ?
2. Comment construire une description d'une application émergente à partir des descriptions unitaires des composants et services ?

### 2.1.3 Feedback de l'utilisateur

Dans notre architecture, le moteur propose des applications qui doivent correspondre aux besoins de l'utilisateur. Pour proposer des applications pertinentes, le moteur OCE a besoin d'enrichir sa connaissance en apprenant le besoin de l'utilisateur [Younes et al., 2019b]. Ainsi, OCE a besoin d'un retour sur les applications qu'il propose tout en ne surchargeant pas son utilisateur en lui demandant une contribution explicite. Le feedback doit donc être

capturé le plus automatiquement possible, en minimisant la sollicitation de l'utilisateur. Pour cela, il peut être construit en se basant sur les modifications que l'utilisateur a fait sur l'application proposée et sur le fait qu'il l'a acceptée ou l'a rejetée.

Les questions de recherche associées à ce problème sont :

1. Comment capturer les préférences de l'utilisateur pour alimenter l'apprentissage d'OCE ?
2. Comment intégrer le moteur avec l'éditeur pour permettre l'échange de données ?

## 2.2 Exigences

Pour répondre aux problèmes posés dans les sections précédentes, nous avons explicité plusieurs exigences listées ci-dessous. D'une manière générale, l'utilisateur doit être informé de l'émergence de nouvelles applications qui sont poussées par OCE. Il doit pouvoir contrôler ces applications grâce à des privilèges qui lui permettent d'éditer, d'accepter ou de rejeter une application. En outre, l'utilisateur doit comprendre ce que l'application lui présente comme service et comment il peut l'utiliser. Enfin, pour améliorer ses décisions, OCE doit apprendre les préférences de l'utilisateur à travers les interventions de ce dernier.

Les exigences sont réparties en 5 catégories : la présentation de l'application, sa compréhensibilité, sa génération automatique, le rôle et les privilèges de l'utilisateur et enfin, le retour d'information de ce dernier.

### 2.2.1 Présentation

Le moteur de composition, selon la disponibilité de composants dans l'environnement à un instant t, propose des assemblages à partir de ces derniers et fait ainsi émerger des applications. Comme ces applications peuvent être inconnues ou inattendues par l'utilisateur, celui-ci doit être informé à la fois de leurs fonctionnalités et de la manière de les utiliser. Pour solliciter a minima l'utilisateur, la présentation doit être automatique.

• **[R1.1] Description fonctionnelle.** La fonction de l'application doit être présentée à l'utilisateur. Par exemple, "L'application permet d'allumer la lampe".

• **[R1.2] Instructions d'utilisation.** Les instructions relatives à l'utilisation de l'application doivent être présentées à l'utilisateur. Par exemple, "Cliquer sur le bouton de l'interrupteur pour allumer ou éteindre la lumière".

### 2.2.2 Compréhensibilité

La présentation de l'application peut être plus ou moins efficace selon le profil de l'utilisateur et ses compétences. Ce dernier peut être un expert en informatique ou un utilisateur moyen qui n'est pas nécessairement familier avec la programmation. Cependant, une bonne compréhension par l'utilisateur des applications présentées est essentielle pour leur acceptation et leur utilisation, et pour la pertinence et la qualité du retour de l'utilisateur. Une autre question est liée à la complexité de l'assemblage en nombre de composants et de services.

• **[R2.1] Intelligibilité de la description.** La description de l'application doit être compréhensible par un utilisateur moyen, sans compétence dans le domaine de la programmation (comme le CBSE par exemple).

• **[R2.2] Extensibilité de la description.** La description doit être utile, compréhensible et intelligible même lorsque l'application comporte plusieurs composants (par exemple, une dizaine de composants).

### 2.2.3 Génération automatique

Le problème réside dans la construction de la description d'une application qui a été construite par une autre entité. Cette construction doit s'effectuer sans intervention ou soutien humain.

• **[R3] Automatisation et composabilité.** La description de l'application doit être construite et présentée automatiquement, en combinant les descriptions unitaires des composants qui la composent et de leurs services.

### 2.2.4 Rôles et privilèges de l'utilisateur

Toutes les applications émergentes, proposées par le moteur, doivent rester sous le contrôle des utilisateurs. C'est à lui de décider de déployer ou de refuser ces dernières. De plus, un utilisateur doit avoir la possibilité de personnaliser lui-même les applications et l'environnement ambiant, en fonction de ses besoins et de ses préférences de configuration. Durant cette tâche, il doit être orienté et guidé pour arriver à des applications pertinentes.

• **[R4.1] Interventions de l'utilisateur.** L'utilisateur doit être capable d'*accepter* les applications émergentes **[R4.1.1]** pour être déployées ou les *rejeter* **[R4.1.2]**. En fonction de son profil et de ses compétences, l'utilisateur doit être capable d'*éditer* et *modifier* le modèle de l'appli-

cation **[R4.1.3]**, c'est-à-dire créer, changer ou supprimer une connexion entre les services des composants qui constituent l'application.

• **[R4.2] Assistance à l'utilisateur.** Durant la phase d'édition, l'utilisateur doit être assisté pour garantir la construction d'applications correctes.

### 2.2.5 Retour d'information de l'utilisateur

La pertinence des applications construites par OCE dépend de sa connaissance du besoin de l'utilisateur. Pour apprendre le besoin de l'utilisateur et être capable de lui fournir les bonnes applications, OCE a besoin de son retour sur les applications proposées. Mais, cette étape doit être faite sans déranger l'utilisateur ni le surcharger par des actions supplémentaires à réaliser [Younes et al., 2019b].

• **[R5.1] Génération du retour d'informations.** Un retour d'information doit être produit pour OCE à partir des interventions de l'utilisateur sur les applications émergentes.

• **[R5.2] Discrétion de la génération du retour d'informations.** L'utilisateur ne doit pas être surchargé par des actions dédiées à la génération du retour d'informations.

Ces exigences ainsi formalisées (sous la forme de 10 énoncés Ri.j) nous ont été utiles pour guider les réflexions et les propositions de solution. De même, les travaux présentés dans l'état de l'art sont analysés à travers le prisme de ces exigences.

# 3 État de l'art

Ce chapitre présente premièrement différents travaux de l'état de l'art dans le but de trouver des solutions qui répondent aux questions de recherche que nous nous sommes posés. Les travaux sont classés dans plusieurs sections où chacune expose un problème.

Enfin, le chapitre présente l'analyse de ces travaux dans 3.5 pour positionner cette thèse par rapport à ces derniers et montrer dans quelle mesure ces travaux répondent aux problèmes identifiés.

## 3.1 L'utilisateur au centre de la boucle

**Designing the Human in the Loop of Self-Adaptive Systems [Gil et al., 2016]** Dans cet article, les auteurs expliquent comment les systèmes auto-adaptatifs, mettant en œuvre le modèle MAPE-K, peuvent se comporter de manière inattendue et possèdent un caractère dynamique et instable. L'auto-adaptation est une exigence clé des systèmes logiciels émergents qui doivent devenir capables d'adapter continuellement leur comportement en cours d'exécution à leur contexte : changement du besoin de l'utilisateur, changement de la disponibilité des ressources et, plus généralement, instabilité de l'environnement.

Ce travail définit les facteurs clés pour favoriser la participation humaine dans les boucles de contrôle en introduisant un cadre pour conserver leur participation : la transparence, l'intelligibilité, la contrôlabilité et la gestion de l'attention des utilisateurs forment ces exigences majeures. Il s'intéresse également à la dynamique entre les différents types de participations humaines en fonction des différentes limitations du système et de la situation actuelle de l'utilisateur.

Les auteurs montrent d'après leurs expérimentations comment l'utilisateur doit être impliqué dans le processus d'adaptation : il peut aider le système à mieux s'adapter aux be-

soins et demandes à travers un apprentissage dynamique de son intervention et de son feedback.

*L'analyse de ce papier et celles des suivants sont présentées à la fin de chapitre (cf. section 3.5).*

**The user in the loop : Enabling user participation for self-adaptive applications [Evers et al., 2014]** Les auteurs montrent que les systèmes informatiques doivent s'adapter de manière autonome aux situations et aux besoins de l'utilisateur (sans le surcharger ou lui demander explicitement). Cependant, le comportement auto-adaptatif n'est pas satisfaisant s'il ne correspond pas aux intentions d'interaction de l'utilisateur.

Les auteurs proposent une solution pour intégrer l'utilisateur dans la boucle, sachant que la convivialité et la modélisation des préférences sont les principales exigences. L'utilisateur sera en mesure d'influencer le comportement d'adaptation au moment de l'exécution et à plus long terme en définissant des préférences individuelles. Pour réussir cette tâche, le système repose sur des modèles de variabilité construits au moment de la conception et des préférences définies par l'utilisateur. Ces processus sont réalisés à travers un outil que les auteurs ont développé dans le cadre de leur projet : MUSIC [Geihs et al., 2009, Floch et al., 2013]. MUSIC n'est pas conçu pour avoir des connexions avec l'application adaptative une fois qu'elle est instanciée et démarrée. Pour la modifier et la reconfigurer, l'application doit être arrêtée puis redéployée.

En outre, pour des raisons d'acceptabilité, les composants "centrés sur l'utilisateur" (c'est-à-dire les composants qui sont dans le champ d'action réel de l'utilisateur, par opposition aux composants "en arrière-plan") sont exclus de l'adaptation dynamique.

Finalement, la contribution de l'utilisateur peut être plus ou moins explicite : il peut sélectionner et ajuster une application, accepter ou rejeter une application, changer ses préférences, ou même repousser le comportement d'adaptation.

**Opportunistic Composition of Human-Computer Interactions in Ambient Spaces [Degas et al., 2016]** Les auteurs montrent que les utilisateurs doivent être maintenus dans la boucle pour interagir avec leurs applications conçues à partir des composants disponibles dans leur environnement ambiant. Ces derniers consistent en un ensemble de dispositifs qui peuvent apparaître et disparaître pour des raisons multiples : mauvais fonctionnement, changement des besoins, déplacement de l'utilisateur dans son environnement... Pour s'adapter à l'instabilité de l'environnement, les applications doivent être conçues d'une manière intelligente en se basant sur l'opportunité d'assembler les composants existants. Ceci mène à un système d'interaction homme/machine (IHM) dynamique.

Pour réussir à maintenir la continuité et le dynamisme des systèmes d'IHM, les auteurs ont développé un système multi-agent qui se base sur l'opportunité de la disponibilité de composants pour assembler des applications. Pour réaliser ces assemblages, chaque composant est géré par un agent qui communique avec d'autres disponibles dans l'environnement pour décider s'ils sont compatibles pour créer une connexion entre eux. Les auteurs ont proposé d'utiliser une "Meta-UI" [Coutaz, 2007] pour présenter les applications émergentes à l'utilisateur et lui permettre de voir les composants utilisés et d'observer l'exécution des applications dans l'environnement. Cependant, l'utilisateur n'a pas un contrôle total sur son environnement et sur ses applications.

**User in the Loop : Adaptive Smart Homes Exploiting User Feedback—State of the Art and Future Directions [Karami et al., 2016]** Dans cet article, les auteurs présentent seulement une architecture générale qui est chargée d'adapter l'automatisation aux différents utilisateurs de leur maison intelligente. L'article se focalise sur la reconnaissance des activités des utilisateurs, leurs feedbacks et l'apprentissage du système. Les auteurs soutiennent que les préférences et le profil de l'utilisateur peuvent être appris (par des algorithmes d'apprentissage de renforcement semi-supervisés), associés à une reconnaissance d'activité qui transforme les données brutes en informations précises sur la situation de l'utilisateur. Ainsi, le feedback de l'utilisateur est un élément important qui aide le système à prendre une décision personnalisée aux besoins de ce dernier dans son environnement et lui proposer des services plus pertinents.

**An adaptive social statistics agent [Isbell et al., 2006]** Les auteurs proposent un système de chat Cobot qui se concentre sur le comportement adaptatif dans des environnements

multi-utilisateurs en utilisant l'apprentissage par renforcement.

Cependant, comme les auteurs le montrent, le problème de l'apprentissage à partir des feedbacks de plusieurs utilisateurs (et non d'experts) doit être traité d'une certaine manière.Les utilisateurs ont des caractères différents et selon leurs besoins actuels, ils ont tendance à soit accepter le service offert à eux ou soit le rejeter. Alors, chaque utilisateur doit posséder une certaine partie du système dédié à son profil qui, à partir de ses préférences et feedbacks, lui propose des services qui répondent à ses besoins.

## 3.2 Développement et programmation par un utilisateur final

**Experience with End-User Development for Smart Homes [Coutaz and Crowley, 2016]** Les auteurs proposent un système qui permet aux utilisateurs finaux de configurer et de contrôler leur maison intelligente à travers une interface. Les habitants de la maison utilisent des langages visuels et pseudo-naturels pour programmer leur environnement ambiant et ajouter ou supprimer des appareils à la volée, sachant que le système les aide par la proposition d'options possibles et les empêche de faire des erreurs.

En conclusion, les auteurs déclarent qu'il est possible de coupler l'EUD avec l'apprentissage. De plus, le service offert aux utilisateurs doit être toujours compris et adaptable à leurs situations.

**Ontology-driven service composition for end-users [Xiao et al., 2011]** Dans cet article les auteurs déclarent qu'il n'existe pas de travaux qui donnent la possibilité aux utilisateurs finaux, ne disposant pas de compétences suffisantes en matière de composition de services, de composer leurs propres services.

Ainsi, les auteurs ont proposé de créer un outil qui assiste les utilisateurs dans la composition du service via un éditeur qui leur permet de préciser leurs objectifs et besoins. Pour créer des applications, le système se base sur des mots-clés introduits par les utilisateurs durant la phase de conception, et en utilisant les descriptions des services et des ontologies disponibles, pour sélectionner les services et les assembler.

Le processus de génération de services possède une architecture ad hoc qui permet aux utilisateurs de personnaliser leurs applications en choisissant des modèles pré-configurés avant de les accepter et de les déployer.

**End User Development : Survey of an Emerging Field for Empowering People [Paternò, 2013]** Dans cet article, les auteurs présentent plusieurs motivations qui favorisent l'EUD. Diverses approches sont examinées et classées en fonction de leurs principales caractéristiques et des technologies et plates-formes pour lesquelles elles ont été développées.

Plusieurs dimensions peuvent être utilisées pour comparer les différentes approches de l'EUD pour les applications.

La première est le caractère générique de l'approche, où il faut vérifier si elle est spécifique à un domaine d'application ou si elle peut être exploitée dans plusieurs domaines en même temps. Par exemple, il existe des approches qui ont ciblé des experts de domaines spécifiques, tels que par exemple des soignants [Carmien and Fischer, 2008], des biologistes [Letondal and Mackay, 2004], des conservateurs de musée [Ghiani et al., 2009]. Dans ces approches des outils permettent aux utilisateurs sans expérience en programmation (mais experts dans leur domaine) de personnaliser les fonctionnalités et l'interface de leurs applications. Cependant, il existe des approches qui ne dépendent pas du domaine d'application et qui ont été utilisées pour divers types d'applications, tel que "App Inventor" [MIT, 2020]. Ce dernier est un environnement qui permet à tous les utilisateurs (experts ou non) de créer des applications entièrement fonctionnelles pour les smartphones.

La deuxième dimension est la couverture des principaux aspects des applications interactives : certaines approches se concentrent uniquement sur le développement de la partie interactive, comme "Denim tools" [Newman et al., 2003], et d'autres visent à couvrir également les parties fonctionnelles, par exemple "ServFace Builder" [Nestler et al., 2010].

## 3.3 Description de services

### 3.3.1 Pour qui et pourquoi les services sont décrits

**Web services composition : A decade's overview. [Sheng et al., 2014]** Les auteurs présentent une vue d'ensemble sur les langages de description de services et les approches de composition automatique de services web. De plus, les auteurs montrent l'utilité de la description des services durant la composition et comment ils sont utilisés.

La composition des services web se déroule sur plusieurs étapes [Zeng et al., 2003] afin de répondre à la demande de l'utilisateur. Tout d'abord l'utilisateur indique son besoin ex-

plicitement, puis la phase de recherche et de sélection de services se déclenche. Cette phase est basée sur la correspondance entre une requête (représentant le besoin de l'utilisateur) et les descriptions des services web disponibles. Il existe deux façons de mettre en œuvre la découverte de services : la correspondance syntaxique et la correspondance sémantique. La première n'offre que des possibilités de recherche basées sur les noms et les identifiants des services. En revanche, dans les méthodes basées sur la sémantique, les services web sont fournis avec des descriptions sémantiques de nombreux aspects (par exemple, les opérations, les entrées, les sorties et même les capacités et le comportement), pour augmenter la précision pour trouver des services web [Wang and Vassileva, 2007, Zeng et al., 2004].

Enfin, après avoir sélectionné les services, le système commence à les assembler pour créer le service composite répondant aux besoins de l'utilisateur [Zeng et al., 2003, Liangzhao Zeng et al., 2004, Sheng et al., 2014].

En conclusion, les descriptions de services sont utilisées par le système pour sélectionner et composer des services qui répondent aux demandes des utilisateurs.

**Ontology-driven service composition for end-users [Xiao et al., 2011]** Les auteurs proposent une plateforme de composition de services centrée sur l'utilisateur, plus précisément sur les utilisateurs finaux sans compétences en ingénierie, pour configurer leurs services. Les utilisateurs finaux définissent d'abord leurs objectifs en utilisant quelques mots-clés. Puis, le système analyse leur demande, cherche et trouve des services pour les assembler afin de créer un ou plusieurs services qui répondent aux besoins des utilisateurs. Ensuite, un éditeur présente les services proposés et suggère des processus possibles et modifiables.

**Semantics Based Service Orchestration in IoT [Chindenga et al., 2017]** Les auteurs présentent une plateforme sémantique de l'internet des objets pour soutenir l'orchestration dynamique des services. Ces derniers peuvent être de plusieurs types et viennent de plusieurs sources hétérogènes.

L'objectif de cette plateforme est d'être capable de proposer à l'utilisateur le service qu'il demande. En outre, la plateforme est basée sur les ontologies pour représenter la sémantique, permettant un format d'échange d'informations cohérent. Ceci rend la phase d'orchestration de services plus pertinente et rend le système capable de sélectionner les meilleurs services répondant à la requête de l'utilisateur. En combinant des ontologies spécifiques à

un domaine, les composants et les services seront décrits d'une manière abstraite, ce qui facilite l'analyse de l'orchestration de services dans des environnements hétérogènes.

**Semantic description of services : issues and examples [Hurault et al., 2009]** Cet article se focalise sur les langages de description de services et montre leur importance dans la composition de services web. Les auteurs présentent plusieurs langages différents dont l'utilisation dépend du contexte du projet. D'une autre manière, il n'existe aucun langage meilleur qu'un autre, mais il faut choisir le langage qui réponde le mieux au problème en question.

Les deux familles de langages présentés par les auteurs sont SOA [Lawler and Howell-Barber, 2007, Newcomer and Lomow, 2005] et OWL [Antoniou and Van Harmelen, 2004, McGuinness et al., 2004].

La première famille décrit les services en se focalisant sur leur signature : types d'entrée et de sortie des données du service, par exemple Java RMI [Oracle, 2020] et WSDL [W3C, 2007]. L'avantage de cette famille de description est la facilité d'accès et de manipulation durant la construction d'un service composite. En revanche, elle n'est pas suffisante pour une composition plus complexe.

La deuxième famille étend les métadonnées du langage et se base complètement sur des ontologies. L'avantage d'utiliser des ontologies est la possibilité d'avoir une description plus formelle et elle permet d'associer plusieurs entités ensemble en se basant sur la description. Ceci permet une sélection de services plus pertinente et plus précise. Cependant, la définition d'une ontologie pour résoudre un problème particulier est difficile à réaliser.

**A Semantically-enhanced Component-based Architecture for Software Composition [Gomez et al., 2006]** L'article présente une architecture pour la composition de composants logiciels en utilisant une description sémantique. Ceci est réalisable à travers l'utilisation d'une ontologie qui définit explicitement la structure conceptuelle des terminologies utilisées dans la spécification des composants. L'objectif de cette architecture est de rendre la composition plus précise et de favoriser la re-utilisabilité des composants dans l'environnement.

### 3.3.2 Comment les services sont décrits

**An overview and classification of service description approaches in automated service composition research [Fanjiang et al., 2017]** Dans cet article, les auteurs présentent et classent les approches de description de services utilisées dans la recherche automatisée sur la composition des services.

En général, tous les langages de description industriels utilisés dans la composition automatisée de services doivent comprendre à la fois des informations fonctionnelles et non fonctionnelles pour décrire un certain élément. Les informations fonctionnelles représentent les signatures d'un service qui consistent en des entrées et des sorties, éventuellement complétées par des conditions [Bartalos and Bieliková, 2012].

Cependant, pour améliorer la description et éventuellement permettre une sélection et une composition de services, la description fonctionnelle est complétée par la description extra-fonctionnelle qui consiste en des caractères d'optimisation et de qualité de service.

Cependant, les langages les plus utilisés dans la recherche sur la composition de services suivent un paradigme conceptuellement simplifié, basé sur des tuples. Ces langages se situent principalement au niveau conceptuel, ce qui signifie qu'ils sont dépourvus de spécifications complexes et ne présentent que les informations requises par l'approche correspondante mise au point par les concepteurs du système.

**Semantic Web Service Description [Klusch, 2008]** Cet article donne une vue d'ensemble sur plusieurs langages de descriptions de services et sur leurs caractéristiques. Une notion intéressante présentée dans cet article est que la description sémantique d'un service peut être seulement fonctionnelle dans la majorité des approches.

En général, la fonctionnalité d'un service peut être décrite à partir de ce qu'il fait et comment il fonctionne réellement. Les deux aspects de sa sémantique fonctionnelle sont saisis par un profil de service et par son modèle de processus.

Le profil décrit la signature du service constituée des paramètres d'entrée et de sortie et de conditions, des effets qui sont censés se produire avant ou après l'exécution du service, ainsi que de certaines informations supplémentaires sur la provenance telles que le nom du service, son domaine d'activité et son fournisseur. Le modèle de processus d'un service décrit comment il fonctionne en termes d'interaction entre les données et de flux de contrôle,

sur la base d'un ensemble commun de flux de travail et sur la communication avec les autres services disponibles.

La sémantique des services non fonctionnels est généralement décrite par rapport à un modèle de qualité de service (QoS) comprenant des contraintes de livraison, un modèle de coût avec des règles de tarification, de répudiation, de disponibilité et de politique de confidentialité. Ces éléments sont définis par les concepteurs et fournisseurs de services.

**Semantic description of services : issues and examples [Hurault et al., 2009]** Dans cet article les auteurs présentent des problèmes dans la description des services et proposent des solutions pour les résoudre. Dans le domaine de composition de services, il existe plusieurs langages qui peuvent être utilisés pour décrire les services de façon à être utilisables durant les phases de composition. Cette dernière consiste en trois étapes majeures : connaissance du besoin de l'utilisateur, recherche et sélection de services et finalement composition.

Cependant, d'après leur état de l'art, les auteurs montrent qu'il n'existe pas de langage idéal. Le premier problème, selon les auteurs, est que c'est difficile de trouver une description suffisamment précise pour pouvoir dire qu'il s'agit d'une description complète d'un service. Deuxièmement, les auteurs soulignent la difficulté de développer des outils qui aident à trouver automatiquement les meilleurs services qui répondent à un certain besoin et les combiner pour créer le service composite.

Par exemple, dans la plupart des architectures orientées service [Newcomer and Lomow, 2005], la composition de services utilise des descriptions qui se basent seulement sur la signature des services (les entrées et les sorties), *p. ex.*, CORBA [OMG, 2020a] et RMI [Oracle, 2020]. Ces types de langages ont l'avantage d'être facilement accessibles et utilisables durant la composition.

Cependant, la signature ne suffit pas car, d'une part différentes fonctions peuvent avoir la même signature et, d'autre part deux services rendant la même fonction peuvent différer fondamentalement dans leurs performances. Par conséquent, la description du service peut inclure des propriétés extra-fonctionnelles, c'est-à-dire des propriétés liées à la qualité de service.

Ainsi, il n'existe pas de solution unique qui puisse convenir à tous les cas, mais, la description doit être adaptée à plusieurs niveaux nécessaires pour répondre aux exigences posées par le contexte : la précision de la description, la rapidité de l'algorithme de correspon-

dance et la gestion de la combinaison.

**Composing web services on the semantic web [Medjahed et al., 2003]** Dans cet article les auteurs proposent un processus de composition de services web qui se base sur la description de services et les ontologies. Les auteurs montrent aussi les différentes techniques pour créer des descriptions de services à partir d'autres qui existent déjà dans l'état de l'art.

Pour créer des langages de description, les auteurs mentionnent que le DAML-S [Ankolekar et al., 2002] est utilisé comme référence de base. Ce langage a introduit les notions de préconditions et de conséquences des services web pour répondre à la composition automatique. Cependant, la façon dont les services composites sont générés à l'aide des spécifications DAML-S n'est pas claire. Le DAML-S ne définit pas la notion de composabilité des services et ne tient pas compte des propriétés sémantiques telles que l'objectif, l'unité de paramètre et le rôle commercial du service.

Actuellement, un langage qui se base sur le DAML-S a été proposé et il est connu sous le nom de OWL-S [Fanjiang et al., 2017, W3C, 2004]. Ce dernier est une ontologie pour la description des services du web sémantique qui permet leur découverte, composition et utilisation automatisées. La description des services basée sur l'ontologie s'est montrée efficace pour la sélection et la composition de services [Xiao et al., 2011].

Une des conclusions des auteurs montre que le choix du langage dépend du contexte de son utilisation. Alors pour une simple composition, un langage basique peut être utilisé. Il est également possible de modifier un langage existant pour répondre à l'exigence du système. Au contraire, si la composition est plus complexe et la qualité de service est importante, les langages qui se basent sur des ontologies sont recommandés.

## 3.4 IDM et systèmes ambiants

**Runtime model based approach to IoT application development [Chen et al., 2015]** Dans cet article les auteurs proposent d'utiliser les technologies de l'IDM pour modéliser les applications Internet des Objets, en anglais "Internet of Things (IoT)", développées dans le but de les configurer avec plus de pertinence. Pour réaliser cette tâche, les auteurs se basent sur une approche divisée en deux parties.

Tout d'abord, pour gérer les dispositifs existant dans l'environnement, les auteurs pro-

posent des créer un métamodèle (avec caractère abstrait) qui représente ces derniers en modèles d'exécution qui sont automatiquement connectés aux systèmes d'exécution correspondants, c'est-à-dire, associer chaque entité différente dans le métamodèle à son système correspondant.

Ensuite, pour être capable de simuler un cas d'utilisation particulier et tester le système, les auteurs proposent de construire un modèle personnalisé correspondant à ce dernier en le synchronisant avec le modèle d'exécution (le métamodèle) défini en avance. Ainsi, l'exécution du modèle sera garantie conforme au scénario personnalisé grâce à l'implémentation de nombreuses méthodes et mécanismes d'analyse ou de planifications centrées sur le modèle et offertes par l'IDM, tels que les vérificateurs de modèles [Rushby, 1995, Combemale et al., 2016]. Ces derniers garantissent que le modèle créé est construit correctement selon le métamodèle, ce qui va permettre d'éviter les erreurs d'exécution.

D'après les expérimentations réalisées, les auteurs montrent l'avantage de l'utilisation d'une telle approche dans le domaine de l'IoT. Premièrement, l'IDM permet une modélisation rapide et à la volée des applications IoT en garantissant la pertinence de cette dernière et en évitant les erreurs de déploiement. Ceci est permis grâce à la définition d'un métamodèle et à la création de modèles personnalisés qui se basent sur ce métamodèle. Deuxièmement, utiliser l'IDM réduit considérablement la charge de travail du codage manuel, car l'ensemble de l'approche ne nécessite que la définition d'un groupe de métamodèles (un par contexte ou domaine d'application) pour manipuler des modèles d'application (initialisés par ces derniers) et générer automatiquement leurs codes.

**MDE4IoT : Supporting the Internet of Things with Model-Driven Engineering [Ciccozzi and Spalazzese, 2016]** Les objectifs pour utiliser les technologies MDE dans les applications IoT sont : (i) permettre de modéliser de façon abstraite les applications pour diminuer la complexité, et (ii) permettre d'automatiser la manipulation des modèles dans la phase d'adaptation à l'exécution. Les avantages sont : 1. MDE facilite la modélisation des applications dans le domaine de l'IoT. 2. C'est plus facile de manipuler et modifier des applications en utilisant les technologies MDE au lieu des technologies génériques. 3. Soutenir les développeurs durant la phase de conception d'une application tout en assurant la cohérence des modèles créés et durant la phase de déploiement grâce à la génération du code par transformation de modèle. 4. MDE permet de faire une adaptation automatique au

changement en temps réel.

Les limites sont : 1. Un système dédié au développeur où l'utilisateur final n'est pas mis dans la boucle, ce qui ne lui permet pas de modifier ses applications pour satisfaire ses besoins. 2. Ils n'ont pas parlé de *model-checking* dans leur approche. Il n'est pas sûr que les applications modélisées soient pertinentes. Ceci est dû au fait de ne pas utiliser un métamodèle spécifique au domaine d'utilisation que souhaite l'utilisateur. En revanche, un métamodèle générique du langage UML est utilisé.

## 3.5 Analyse et positionnement

Le tableau 3.1 (p. 50) présente l'analyse des travaux de l'état de l'art en montrant comment chacun d'eux peut répondre aux exigences que nous avons définies. Le taux de réponse à une exigence varie entre zéro et quatre "+". Dans ce tableau, lorsqu'une exigence n'est pas couverte, cela signifie qu'elle est notée 0. De plus, les articles qui ne possèdent aucune réponse à (au moins) une exigence sont des articles de synthèse de plusieurs travaux de composition ou sur des langages de description existants.

D'après cette analyse, aucun travail existant ne répond explicitement aux exigences [R1.1], [R1.2] et [R2.2]. Dans la majorité des travaux, les descriptions de services sont utilisées comme documentation, qui détaillent leur intention et leur utilisation. Une fois que l'utilisateur a défini son besoin, les concepteurs du système ou les processus automatisés de composition, en utilisant une approche "Top-down", sélectionnent les meilleurs services pour créer l'application [Chindenga et al., 2017, Gomez et al., 2006, Fanjiang et al., 2017, Klusch, 2008]. La sélection des services se base sur leur description pour choisir les services qui répondent au mieux au besoin de l'utilisateur. Dans certains travaux, comme [Chen et al., 2015, Ciccozzi and Spalazzese, 2016], les applications créées sont présentées par les concepteurs à l'utilisateur en lui montrant ses fonctionnalités. Ceci n'est pas possible dans notre projet puisque la conception des applications est réalisée par une entité logicielle, le moteur de composition.

Traditionnellement, les services composites demandés sont spécifiés au début de manière à répondre à un besoin défini explicitement [Gil et al., 2016, Evers et al., 2014, Xiao et al., 2011], de sorte qu'il est inutile de produire des descriptions par la suite, puisque la fonctionnalité du service proposé est connu. Ce n'est pas le cas dans la composition op-

portuniste qui n'est pas guidée par un besoin explicite. Dans cette dernière, le moteur de composition ne se base pas sur la description des services pour choisir les meilleurs et produire des applications, mais il se base sur les préférences de l'utilisateur qui sont apprises au fur et à mesure de l'exécution du système. Nous n'avons pas trouvé de travaux qui s'intéressent à la génération d'une description d'une application émergente. Ainsi, il n'existe pas de solutions utilisables pour répondre à tous les critères de [R1.1], [R1.2] et [R2.2].

Pour ce qui concerne les exigences [R2.1], [R4.1] et [R4.2], il existe des travaux qui ont cherché à y répondre en proposant des systèmes dédiés aux profils de leurs utilisateurs. En créant des éditeurs *ad hoc*, les auteurs ont réussi à présenter à l'utilisateur le résultat sous forme d'une représentation compréhensible et à lui permettre, d'une certaine manière, de faire des modifications possibles avant de déployer l'application. Dans [Coutaz and Crowley, 2016] les auteurs ont proposé un éditeur *ad hoc* qui permet aux utilisateurs de différents profils (mais possédant un certain niveau en programmation) de programmer leur environnement. Par exemple, les habitants de la maison utilisent des langages visuels et pseudo-naturels pour personnaliser leur environnement ambiant. En outre, [Evers et al., 2014] proposent un éditeur qui présente à l'utilisateur plusieurs applications potentielles : il peut sélectionner et ajuster une application en changeant quelques éléments, accepter ou rejeter une application, voir même changer ses préférences.De la même manière, dans [Xiao et al., 2011] les auteurs proposent un éditeur dédié aux utilisateurs moyens pour les assister pendant la phase de conception de leurs applications. En précisant son besoin, l'utilisateur obtient une application avec une liste de modifications possibles pour lui permettre d'éventuellement changer la conception de cette dernière.

Cependant, dans les travaux qui répondent à ces exigences, certaines compétences sont requises par l'utilisateur, qui ne peut donc pas être novice. L'utilisateur ne possède pas un contrôle total sur la conception et il est limité à certaines modifications possibles sur son application qui n'est conçue qu'après avoir explicitement formulé les besoins. Ainsi, il y a peu ou pas d'intelligence (et pas d'émergence) dans les solutions EUD existantes, qui n'offrent que très peu de flexibilité et de personnalisation des application contrairement à la solution souhaitée dans cette thèse.

L'exigence [R3] est celle pour laquelle plusieurs solutions et approches sont proposées. Que ce soit dans le domaine de composition web [Gomez et al., 2006, Klusch, 2008] ou dans le domaine d'application logicielle [Isbell et al., 2006, Coutaz and Crowley, 2016,

Degas et al., 2016], les auteurs proposent des systèmes autonomes qui offrent des applications et des services à leurs utilisateurs.

Néanmoins, l'automatisation ne porte pas uniquement sur le processus de composition et de proposition d'une application mais aussi sur la réalisation d'une description de cette application. Or, les travaux de l'état de l'art ne prennent pas en considération ce point, surtout que la plupart des travaux se basent sur des préférences pré-définies par l'utilisateur pour sélectionner les composants optimaux et les assembler en une application.

Par ailleurs, il existe des approches telles que [Chen et al., 2015] et [Ciccozzi and Spalazzese, 2016] où les auteurs ont proposé des approches qui se basent sur l'IDM pour la conception, la présentation et le déploiement d'applications. Ces derniers décrivent les avantages des technologies de l'IDM et comment ces dernières facilitent la présentation des applications, réduisent la charge de codage, rendent le système plus automatisé et facilitent la manipulation des modèles. Ceci nous a motivé pour utiliser les technologies de l'IDM notamment pour la manipulation des modèles d'assemblage construits par OCE.

En outre, ces approches répondent aussi à [R2.1], [R4.1] et [R4.2], à travers les éditeurs spécifiques développés pour modéliser les applications, les manipuler et ensuite générer leur implémentation. Cependant, les utilisateurs de ces éditeurs sont des concepteurs d'application et non pas, comme nous, des utilisateurs finaux. Ces solutions ne pourront donc pas être utilisées telles quelles, mais des solutions d'IDM peuvent être développées en modifiant la manière de présenter et de manipuler des modèles.

Il existe plusieurs travaux qui s'intéressent à la question de l'identification des besoins de l'utilisateur à travers son feedback, répondant ainsi partiellement aux exigences [R5.1] et [R5.2]. Dans [Gil et al., 2016], les auteurs montrent qu'à travers un apprentissage dynamique des interventions de l'utilisateur, le système peut s'adapter à son besoin actuel. Dans [Karami et al., 2016], les auteurs montrent que, d'après leur architecture, l'apprentissage du feedback est nécessaire pour aider le système à proposer des services personnalisés. En outre, les auteurs de [Isbell et al., 2006] montrent que dans les environnements multi-utilisateurs ceux-ci possèdent des profils différents et que, par conséquent, leurs besoins varient énormément. L'apprentissage doit donc apprendre et différencier les besoins en fonction des types d'utilisateurs pour proposer des applications pertinentes personnalisées. Ces travaux confirment la nécessité d'implémenter un mécanisme d'apprentissage des besoins

de l'utilisateur. Cet apprentissage ne concerne pas directement les travaux de cette thèse : c'est OCE qui modifie les connaissances des agents pour décider des prochaines connexions en fonction des préférences de l'utilisateur. Cependant pour cela, il a besoin de recevoir un retour sur l'application qu'il vient de proposer. Ainsi, pour répondre à cette exigence [R5.1], il faut, d'après les interventions de l'utilisateur, identifier et générer les données nécessaires à envoyer au moteur de composition pour son apprentissage. En outre, cette extraction de feedback doit être la plus discrète et légère possible pour ne pas surcharger l'utilisateur. Nous n'avons pas trouvé de travaux qui répondent totalement à [R5.2].

| | Points forts | Points faibles | Réponse aux Exigences |
|---|---|---|---|
| [Gil et al., 2016] | Auto-adaptation aux changements de l'environnement<br>Capture des interventions de l'utilisateur | Pre-analyse des exigences de l'utilisateur<br>Proposition sous forme d'une conception (aucun prototype) | [R4.1] : + [R5.1] : ++ [R5.2] : + |
| [Evers et al., 2014] | Adaptation aux besoins de l'utilisateur<br>Editeur favorisant les interventions de l'utilisateur | Pre-définition des besoins de l'utilisateur<br>La modification des applications est limitée aux exemples proposés par le système | [R4.1] : + [R4.2] : + [R5.1] : ++ |
| [Degas et al., 2016] | Adaptation aux imprévus de l'environnement ambiant | Présentation des applications sans donner le pouvoir à l'utilisateur de les modifier<br>Système non basé sur le feedback de l'utilisateur | [R5.1] : + [R5.2] : + |
| [Karami et al., 2016] | Adaptation aux besoins de l'utilisateur<br>Apprentissage via le feedback | Présentation d'une architecture du système seulement, sans aucune mise en œuvre | [R5.1] : ++ [R5.2] : + |
| [Isbell et al., 2006] | Apprentissage du besoin de l'utilisateur via le feedback<br>Système dédié à des utilisateurs dont les profils sont différents | Plusieurs systèmes dédiés à chaque utilisateur<br>L'utilisateur n'est pas l'acteur principal | [R2.1] : ++ [R4.1] : + [R5.1] : ++ [R5.2] : + |
| [Coutaz and Crowley, 2016] | Configuration des applications via un éditeur<br>Les utilisateurs sont assistés durant la phase d'édition | Feedback de l'utilisateur non pris en considération<br>Système dédié à des utilisateurs experts | [R4.1] : ++ [R4.2] : ++ |
| [Xiao et al., 2011] | Composition de service qui se base sur le besoin de l'utilisateur<br>Assistance de l'utilisateur lors de la configuration de son service | Pré-définition des besoins de l'utilisateur<br>La modification des applications est limitée aux exemples proposés par le système<br>Architecture ad hoc dédiée à un seul cas d'utilisation | [R2.1] : ++ [R4.1] : + [R4.2] : + [R5.1] : + |
| [Sheng et al., 2014] | Importance de la description durant la phase de sélection et composition de services | Pré-définition du besoin d'une manière explicite<br>Aucun privilège donné à l'utilisateur pour modifier le service proposé | [R5.1] : + [R5.2] : + |
| [Chindenga et al., 2017] | Composition automatique des services<br>Amélioration de la pertinence des applications grâce aux ontologies | Aucune référence sur comment le système peut apprendre le besoin de l'utilisateur<br>L'utilisateur ne peut faire aucune intervention sur le service proposé | [R2.1] : + [R4.1] : ++ [R4.2] : + |
| [Chen et al., 2015] | Utilisation d'une approche IDM pour une configuration<br>plus pertinente des applications IoT<br>Réduction de la charge de codage et obtention d'un système plus automatisé | Définition d'un métamodèle pour chaque cas d'utilisation<br>Application créée par le concepteur puis présentée à l'utilisateur final | [R2.1] : ++ [R4.1] : + [R4.2] : + [R5.1] : + |
| [Ciccozzi and Spalazzese, 2016] | Diminution de la complexité du système<br>Automatisation de la manipulation des modèles d'application | Système non dédié à l'utilisateur final<br>La construction des applications n'est pas validée par des métamodèles pré-définis | [R2.1] : ++ [R4.1] : + [R4.2] : + [R5.1] : + |

Table 3.1 – Réponses des travaux étudiés aux exigences requises

*Deuxième partie*

# Description automatisée centrée utilisateur des applications composites émergentes

# 4 Approche basée sur l'IDM pour assister l'utilisateur dans son environnement

Afin de mettre l'utilisateur au centre de la boucle et de répondre au problème de la présentation multivue, et pour satisfaire les exigences exposées, un environnement de contrôle interactif est proposé, appelé ICE (*Interactive Control Environment*). ICE fonctionne en collaboration avec le moteur de composition opportuniste OCE.

Le développement d'ICE est basé sur des technologies et principes de l'IDM. Il consiste à définir le métamodèle, un DSL et les transformations de modèle nécessaires pour présenter l'application, favoriser les interactions de l'utilisateur et produire son feedback.

Ce chapitre présente globalement notre solution et ses principes. Les sections 4.1 et 4.2 donnent une vue d'ensemble : éléments responsables de la présentation de l'application et outils permettant l'intervention de l'utilisateur (les détails du développement sont donnés dans la partie III "Développement, expérimentation et analyse"). La section 4.3 introduit le processus de capture de feedback destiné à l'apprentissage du moteur OCE. Enfin, la section 4.4 expose nos motivations pour l'utilisation des technologies de l'IDM.

Les chapitres qui suivent présentent en détail les différents éléments de cette contribution, puis analysent ses avantages et ses limites.

## 4.1 L'environnement de contrôle interactif ICE

La figure 4.1 montre une vue d'ensemble de l'outil proposé.

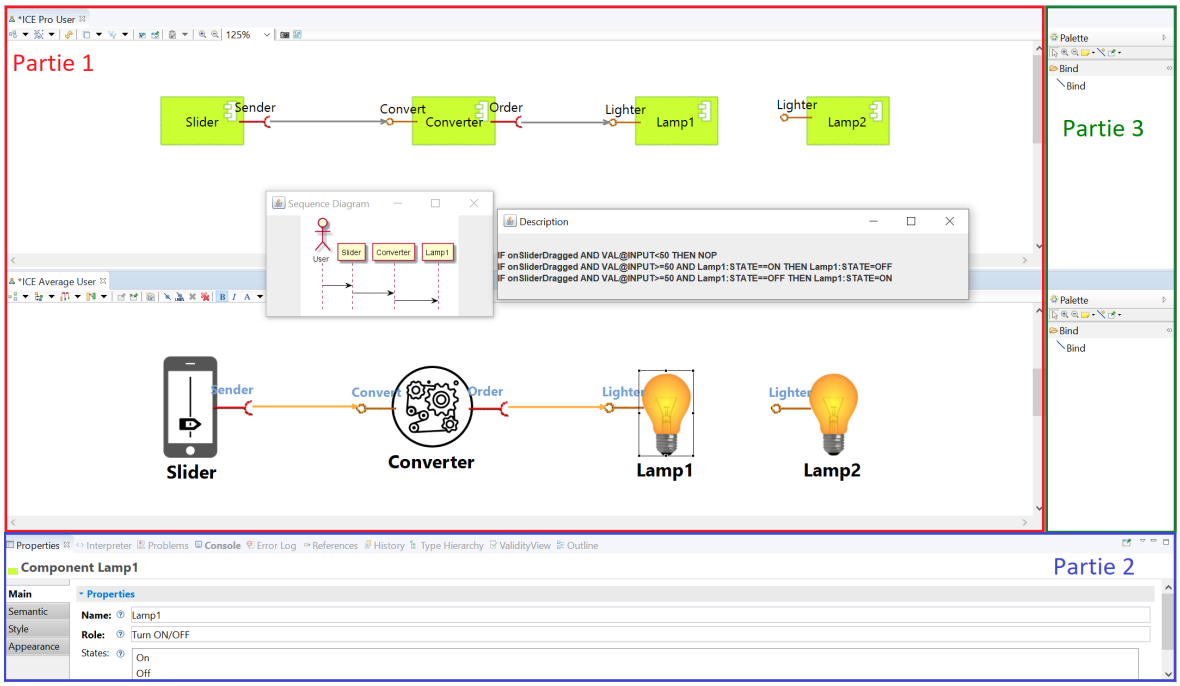


Figure 4.1 – Environnement de Contrôle Interactif (ICE)

ICE est un éditeur graphique qui permet à l'utilisateur de contrôler l'environnement et les applications qui émergent de ce dernier. Pour réaliser cette tâche, ICE présente les modèles des applications proposés par OCE. L'utilisateur peut ainsi connaître leur structure et décider s'il veut faire des modifications et procéder à l'acceptation et au déploiement, ou simplement rejeter l'application.

L'éditeur graphique se divise en trois parties. La partie 1 permet l'affichage des applications émergentes proposées par OCE sous plusieurs formes. L'utilisateur, en fonction de ses compétences en programmation, peut voir la structure de l'application sous la forme d'un diagramme de composants UML [OMG, 2020b] ou sous la forme d'un diagramme à base d'icônes. Cette dernière forme, qui présente l'application d'une manière plus expressive et compréhensible, est plus pertinente qu'un diagramme de composants pour des utilisateurs non informaticiens. Ceci répond à l'exigence [R2.1].

Dans le diagramme de composants, les composants non participants à l'application sont également affichés : ce sont des candidats à la composition que l'utilisateur peut utiliser, durant la phase de modification, pour remplacer certains composants de l'application initiale.

En outre, à travers cette première partie de l’interface, l’utilisateur peut accepter ou rejeter l’application proposée. Ceci répond aux exigences [R4.1.1] et [R4.1.2].

De plus, les utilisateurs peuvent bénéficier d’autres vues de l’application qui montrent d’une manière plus explicite ses fonctionnalités et son mode d’utilisation : un diagramme de séquence et une description à base de règles. Ceci est une réponse aux exigences [R1.1] and [R1.2].

Toutes ces représentations sont construites et générées automatiquement par ICE sans intervention de l’utilisateur, ce qui répond à [R3].

Par ailleurs, ICE offre à l’utilisateur un service de support à l’édition des applications. La partie 2 de l’interface affiche les différentes propriétés des composants et de leurs services. Pour effectuer des modifications, l’utilisateur dispose dans la partie 3 d’un outil qui lui permet de créer de nouvelles connexions entre les services des composants affichés. Le processus d’édition ne consiste pas seulement à créer des connexions car l’utilisateur peut aussi supprimer des connexions existantes ou les modifier en changeant l’un des services connectés : dans l’exemple de la figure 4.1 l’utilisateur, habitant d’un appartement connecté, peut remplacer la connexion du service *Order* du composant *Converter* par une connexion au service *Lighter* du composant *Lamp2*, et ceci en cliquant sur la connexion et en la faisant glisser sur la nouvelle destination. Ceci répond aux exigences [R4.1.3] et [R4.2].

## 4.2 Architecture “Utilisateur au centre de la boucle”

ICE est un élément de l’architecture globale du système de composition qui place l’utilisateur au centre de la boucle du système. Une vue d’ensemble de cette architecture est présentée dans la figure 4.2. La solution globale est conçue et réalisée en collaboration avec un autre doctorant de l’équipe, Walid Younes, qui travaille sur la partie OCE.

Les flèches rouges détaillent le flux de données entre différentes entités du système. Les polygones (Décision, Apprentissage, Présentation, Édition, Acceptation et Capture du Feedback) représentent les différentes étapes par lesquelles passe l’application depuis son émergence jusqu’à son déploiement.

Après avoir détecté les composants disponibles dans l’environnement, OCE, en se basant sur sa connaissance, propose une application qui consiste en un plan d’assemblage de composants. Cette dernière est présentée à l’utilisateur via un éditeur graphique qui lui

permet de visualiser les différents composants utilisés. Ceci répond à l'exigence [R3]. L'utilisateur peut ensuite la modifier si besoin et finalement l'accepter pour la déployer dans l'environnement ou la rejeter. Ceci répond à [R4.1].

Enfin, les interventions de l'utilisateur sont capturées pour construire le feedback qui sera envoyé à OCE. Ce dernier peut ainsi apprendre le besoin de l'utilisateur et mettre à jour sa connaissance.

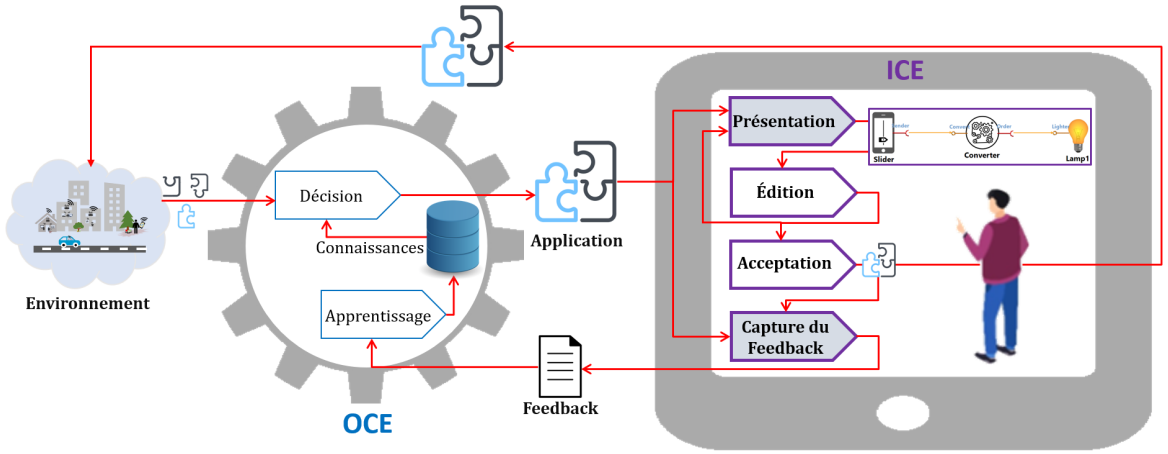


Figure 4.2 – Vue d'ensemble de l'architecture

## 4.3 Retour d'information sur l'utilisateur

Pour répondre à l'exigence qui concerne le retour d'information pour l'apprentissage ([R5.1]), les actions et les interventions de l'utilisateur peuvent être capturées via l'interface, puis enregistrées par ICE pour être transformées en un *modèle de feedback* qui sera envoyé à OCE. Ce modèle contient l'historique des différentes modifications réalisées par l'utilisateur sur l'application originalement proposée par OCE.

À sa réception, OCE utilise ce modèle pour apprendre le besoin de l'utilisateur et mettre à jour sa connaissance. Ceci va permettre à OCE de proposer plus tard des applications plus pertinentes et adaptées à l'utilisateur. Avec ce choix d'apprentissage *a posteriori*, l'utilisateur n'est pas contraint de fournir explicitement ses besoins. L'exigence [R5.2] est donc satisfaite.

## 4.4 Motivations pour utiliser l'IDM

Pour mettre en oeuvre notre proposition, nous avons exploité les technologies offertes par l'IDM (cf. section 1.4). Plusieurs observations peuvent être faites pour justifier ce choix :

1. Fondamentalement, OCE produit un modèle d'application émergente c'est-à-dire un plan d'assemblage de composants. Ces modèles sont tous conformes au même métamodèle et consistent en un ensemble de composants et un ensemble de connexions entre leurs services. Le métamodèle est prescriptif car il précise l'application à créer. Il est aussi descriptif en décrivant l'application lorsqu'elle est acceptée et déployée [Ludewig, 2003].

2. Les modèles d'application produits par OCE doivent être transformés et présentés à l'utilisateur à travers plusieurs formes : la forme des modèles de description de l'application est fonction du profil de l'utilisateur et de son niveau de connaissances informatiques. Par exemple, la présentation sous la forme d'un diagramme de composants (structurel) permet à un utilisateur, possédant des compétences en programmation, d'effectuer si nécessaire la reconfiguration de l'application, voire la configuration d'une nouvelle.

3. Durant la phase d'édition, l'utilisateur doit être assisté par des fonctionnalités de modification et des règles de contrôle pour lui permettre de créer des applications pertinentes (applications valides) conforme au métamodèle (par exemple, sans faire des connexions non autorisées). Comme les modèles peuvent être modifiés par les utilisateurs, ils peuvent également être qualifiés de modèles exploratoires [Ludewig, 2003].

Sur la base des éléments exposés ci-dessus, nous nous sommes donc orientés vers une approche basée sur l'IDM pour développer et supporter ICE. Cette approche doit permettre de modéliser et de présenter à l'utilisateur les applications émergentes grâce aux méthodes de manipulation de modèle offertes par l'IDM plutôt que d'utiliser des approches ad hoc. Elles permettent également de transformer le modèle d'application accepté par l'utilisateur, éventuellement après modification, en un modèle de données constituant le feedback pour l'apprentissage d'OCE et de générer un script de déploiement de l'application.

Ce mode d'utilisation de l'IDM est original. En général, les outils et les techniques de l'IDM sont conçus pour être utilisés par des ingénieurs experts afin de modéliser et de développer un logiciel et de générer le code pour son implémentation. Ici, l'IDM cible les uti-

lisateurs finaux qui possèdent des profils et des niveaux de compétences différents (utilisateurs moyens ou experts). Outre la transformation des modèles des applications émergentes, l'IDM supporte la production d'éditeurs que les utilisateurs peuvent utiliser pour faire les modifications souhaitées sur les applications.

Les chapitres suivants décrivent en détail les éléments de notre contribution. Le chapitre 5 explique comment une application est présentée à l'utilisateur sous forme d'une représentation structurelle. Le chapitre 6 discute de la production de représentations plus riches, plus détaillées, qui explicitent davantage la sémantique des applications. Enfin, le chapitre 7 décrit les privilèges accordés à l'utilisateur pour intervenir tout en étant assisté, et comment le feedback est généré afin d'être transmis à OCE.

# 5 Présentation structurelle de l'application émergente

Les systèmes ambiants, avec leur caractère dynamique et leur imprévisibilité, ne permettent pas de concevoir, développer et mettre en œuvre des applications en suivant les cycles de développement traditionnels. Cependant, il faut implémenter des infrastructures qui permettent de réaliser, présenter et déployer automatiquement les applications [Lewis and Fowler, 2014]. Notre projet vise à aller plus loin dans cette direction en automatisant l'assemblage des composants qui sont disponibles dans l'environnement et en faisant émerger des applications opérationnelles. Dans ce contexte, l'utilisateur doit être mis dans la boucle pour décider de l'intérêt et de l'utilisation des applications émergentes.

L'utilisateur doit donc toujours être informé des applications qui sont découvertes par OCE. Pour cela, lorsqu'une application émerge de l'environnement, elle est présentée à l'utilisateur. La forme de cette présentation doit permettre de montrer les composants présents dans l'assemblage ainsi que les connexions entre services. Ceci est un premier pas pour permettre à l'utilisateur de comprendre l'application. En résumé, il s'agit donc de répondre au problème suivant : comment, à partir d'une liste de connexions entre services de composants logiciels (description d'un assemblage construit par OCE), arriver à une présentation visuelle lisible et compréhensible pour un utilisateur ?

Nous utilisons une approche basée sur l'IDM pour présenter une application sous forme d'un diagramme de composants UML [OMG, 2020b] grâce à un éditeur. Cette approche consiste à définir un métamodèle d'assemblage et ensuite un DSL pour visualiser l'application. L'utilisation des méthodes de transformation de modèle s'est avérée nécessaire pour traduire le modèle d'assemblage d'OCE en un modèle compatible avec l'éditeur graphique.

Ce chapitre commence par présenter le métamodèle proposé qui définit un assemblage de composants dans un environnement. Puis sont présentés, le DSL permettant la présenta-

tion de l'application sous forme d'un diagramme de composants, ainsi que la transformation de modèle nécessaire pour obtenir le modèle de l'application. Le chapitre se termine par une analyse des propositions et présente la limite de cette contribution.

## 5.1 Métamodèle d'un assemblage de composants

Plusieurs métamodèles de composants et de services existent dans l'état de l'art et peuvent être utilisés pour représenter un assemblage de composants. Cependant, ces derniers présentent une application émergente d'une façon trop complexe pour les besoins d'ICE.

Par exemple, dans [Tigli et al., 2009], les auteurs proposent un métamodèle de composition de composants en introduisant la notion d'architecture de composition de services web. Ce dernier décrit ainsi le numéro du port d'un service, ses paramètres, ses événements, ses méthodes et son lien d'implémentation. Il indique aussi le chemin d'accès pour récupérer un composant d'un dépôt de composants en ligne. Ce métamodèle décrit donc les composants et leurs interfaces (services) à un niveau de détail non nécessaire dans notre cas.

[Janisch, 2010] propose un métamodèle (similaire au précédent) qui se base aussi sur la notion de composition web mais en détaillant la notion d'assemblage. Ce dernier décrit la réalisation d'un assemblage en introduisant la notion de connexion synchrone et asynchrone, là encore, inutile pour nous.

Dans [Hahn and Fischer, 2007] le métamodèle se base sur approche multi-agent pour la réalisation d'assemblages dans les architectures orientées services (SOA) [Lawler and Howell-Barber, 2007, Newcomer and Lomow, 2005]. Il modélise les différentes caractéristiques des agents : leur comportement, leur rôle et leur organisation pour réaliser une composition. De plus, ce métamodèle décrit comment les agents échangent leurs messages et les protocoles utilisés.

Comme ces métamodèles se basent sur les SOA et se focalisent plus sur la réalisation d'une application que sur sa présentation à un utilisateur, ils ne répondent pas vraiment à nos exigences. Aussi, nous avons préféré définir un métamodèle dédié à notre proposition, conformément à ce qui a été introduit dans la section 1.1. Ce dernier est présenté dans la figure 5.1.

Le métamodèle d'un assemblage consiste en un environnement (représenté par la classe

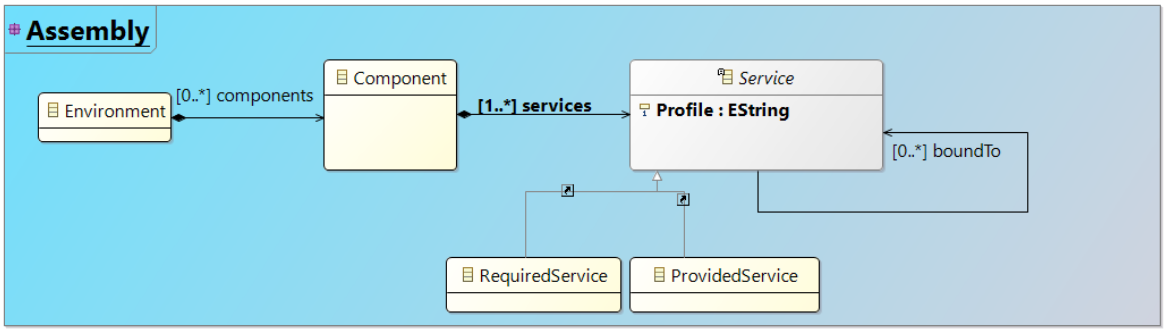


Figure 5.1 – Métamodèle d'un assemblage de composants

*Environment*) qui peut contenir des Composants (représentés par la classe *Component*). Les composants dans un environnement dynamique peuvent apparaître ou disparaître. Leur nombre est donc variable, éventuellement égal à 0. Ceci est pris en compte avec la relation $[0..*]$ entre les classes *Environment* et *Component*.

Pour être composable un composant doit avoir au moins un service. Ceci est représenté par la relation $[1..*]$ entre les classes *Component* et *Service*.

Un service peut être de deux types, requis ou fourni. Ceci est représenté par les relations d'extension entre les classes *RequiredService*, *ProvidedService* et la classe abstraite *Service*. Un service possède une propriété nommée *boundTo* qui définit les connexions entres les services lorsque l'application est créée.

La classe *Service* possède aussi l'attribut *Profile*. Ce profil, défini pour chaque service, est utilisé pour ajouter des contrôles sur le processus de modification de l'application (cf. chapitre 7). Par exemple, cet attribut contient la signature d'un service qui est utilisé pour réaliser les connexions. En outre, il est utilisé pour garantir que seuls les services compatibles peuvent être connectés ensemble et que le nombre maximal de connexions d'un service est respecté.

De plus, des règles OCL sont définies pour mettre des restrictions sur le modèle des applications. La figure 5.2 montre un exemple de règles OCL qui vérifie que deux services fournis (1ère règle) ou que deux services requis (2ème règle) ne peuvent pas se connecter ensemble.

```
invariant checkBoundToRequired : self.boundTo -> forAll(s1 | s1.oclIsKindOf(RequiredService));
invariant checkBoundToProvided : self.boundTo -> forAll(s1 | s1.oclIsKindOf(ProvidedService));
```

Figure 5.2 – Exemple de règles OCL définies sur le métamodèle

Notons que le métamodèle d'assemblage présenté dans la figure 5.1 fait partie d'un métamodèle plus complexe présenté dans le chapitre suivant. Le métamodèle complet supporte la présentation de l'application à l'utilisateur mais aussi la génération de la description de cette application.

## 5.2 DSL pour une représentation sous forme d'un diagramme de composants

ICE présente des modèles d'assemblage de composants que les ingénieurs logiciels réalisent généralement sous forme de diagrammes de composants UML. Pour faire cette représentation de modèles de type UML, un DSL a été défini et est conforme au diagramme de composants UML. La figure 5.3 montre un exemple d'une représentation d'un assemblage en utilisant le DSL de diagramme de composants.


Figure 5.3 – Représentation du diagramme de composants de l'application pour un utilisateur expert via notre DSL

Ce DSL contient une définition de chaque représentation visuelle des éléments du métamodèle définie en 5.1. Les composants sont représentés par un rectangle vert, les services requis sont représentés sous forme d'un demi-cercle, les services fournis sont sous la forme d'un cercle complet et la propriété "boundTo" par une flèche grise.

## 5.3 Transformation du modèle OCE en modèle ICE

Pour être présenté, le modèle de l'application généré par OCE doit être transformé en un modèle compréhensible, manipulable et modifiable par l'utilisateur. Cette transformation est une exigence conséquente aux [R1.1] et [R1.2]. Sachant que les modèles d'OCE et d'ICE sont conformes au même métamodèle d'assemblage (voir figure 5.1), la transformation requise ici est de type modèle-à-modèle. Plus précisément, le type de cette transformation, où le modèle généré par OCE est transformé en un modèle ICE, est endogène.

Une fois généré, le modèle ICE est injecté dans l'éditeur graphique pour être présenté à l'utilisateur (grâce au DSL présenté en section 5.2). Il contient les différents composants avec leurs services connectés et leurs attributs. Il peut aussi contenir des composants candidats

disponibles à ce moment là dans l'environnement qui peuvent servir à l'utilisateur pour modifier l'assemblage (cf. chapitre 7). Par exemple, la figure 5.3 montre une application avec trois composants connectés ainsi qu'un composant candidat Lamp2 qui pourrait remplacer Lamp1 dans l'assemblage.

## 5.4 Analyse de la proposition

Nous avons expliqué dans ce chapitre comment, grâce aux outils et aux principes de l'IDM, notre système présente des applications émergentes à l'utilisateur. Ce dernier peut ainsi visualiser les composants qui constituent l'application, leurs connexions ainsi que les composants candidats (qui peuvent être utiles si l'utilisateur souhaite modifier l'application).

La contribution de ce chapitre ne porte pas sur les modèles ou outils de l'IDM proprement dits mais réside dans l'originalité de leur utilisation puisqu'elle vise l'utilisateur final. Par rapport à l'état de l'art, il n'existe pas de travaux qui utilisent l'IDM pour créer des systèmes dédiés à l'utilisateur final, au contraire les approches de l'IDM sont dédiées aux ingénieurs pour concevoir et réaliser des applications.

Cependant, pour comprendre un diagramme de composants, l'utilisateur doit posséder des compétences en informatique. Par conséquent, un utilisateur non informaticien risque de rencontrer des difficultés pour comprendre la fonctionnalité et l'utilisation d'une application émergente. Alors, cette proposition possède une limite en terme d'intelligibilité lors de la présentation de l'application à l'utilisateur.

Par conséquent, une autre contribution a été proposée. Elle consiste à présenter également l'application sous d'autres formes de descriptions multivues, plus riches et plus détaillées, dédiées aux utilisateurs moyens. Les descriptions multivues sont présentées dans le chapitre suivant.

# 6 Description multivue de l'application émergente

Nous avons présenté une architecture qui met l'utilisateur au centre de la boucle. L'éditeur est capable de présenter l'application, sous forme d'une représentation structurelle en montrant les différents composants qui sont connectés par leurs services respectifs, ainsi que d'autres composants disponibles qui peuvent être utiles. Ceci est réalisé en utilisant des techniques de l'IDM qui transforment la sortie du moteur en un modèle présentable à l'utilisateur. Cette proposition permet à l'utilisateur de modifier, d'accepter ou de rejeter l'application, mais elle ne l'aide pas à comprendre la sémantique de cette dernière ou le service offert. La fonction et la manière d'utiliser une application sont pourtant importantes et utiles pour décider d'utiliser ou non une nouvelle application.

Aussi, nous présentons dans ce chapitre les contributions qui visent à présenter une application de façon plus riche et détaillée grâce à des formes de représentation différentes et des descriptions sémantiques. Ces différentes vues permettent à un utilisateur (surtout s'il n'est pas familier avec la programmation) de mieux comprendre l'application émergente. Elles l'aideront à décider de ce qu'il veut faire de cette application.

Le reste de ce chapitre est organisé comme suit. Tout d'abord les motivations qui nous ont amenées à enrichir les formes de représentation sont présentées. Ensuite, chaque élément de contribution est détaillé : la proposition d'un deuxième DSL graphique ainsi que les différentes transformations de modèle pour obtenir par exemple une description de l'application à base de règles. Enfin, le chapitre se termine par une analyse des propositions de représentation multivue d'une application émergente.

## 6.1 Introduction

Dans le chapitre précédent, nous avons utilisé l'IDM pour passer du modèle construit par OCE à une description UML plus explicite. Cependant, cette présentation de l'application sous la forme d'un diagramme de composants requiert quand même certaines compétences en informatique pour la comprendre. Ainsi, sur la seule base de cette description, un utilisateur moyen pourrait être incapable de connaître comment son application a été conçue ou ne pas savoir quel service elle lui offre.

L'idée est alors, toujours avec les technologies de l'IDM, de présenter l'application, sous la forme de plusieurs vues qui "parleraient" à un utilisateur moyen. Le rôle de ces vues seraient de présenter l'application d'une façon plus intelligible et de décrire son caractère sémantique et opérationnel. Ces nouveaux types de description sont rendus possibles, d'une part par la définition d'un nouveau DSL graphique pour l'utilisation de pictogrammes et d'autre part, en utilisant des méthodes de transformation de modèle pour générer à partir du modèle de l'application différentes vues alternatives (un diagramme de séquence ou une description à base de règles).

## 6.2 DSL à base de pictogrammes

Comme mentionné précédemment, l'utilisateur peut ne pas être familier avec une représentation sous la forme de diagramme de composants, ce qui va l'empêcher de comprendre la structure de l'application. Dans [Abrahão et al., 2017], les auteurs prétendent qu'une boîte dans laquelle est écrit “Cat” (dans notre cas “Lamp1”) peut être comprise par un ingénieur logiciel comme représentant un chat, mais que cela ne reste qu'une "boîte" pour la plupart des gens (un composant logiciel dans notre cas). Pour résoudre ce problème, le DSL du diagramme de composants peut être adapté pour personnaliser la présentation, en remplaçant la représentation UML du composant par une icône plus explicite pour le grand public. Un second DSL a donc été défini. De cette manière, selon les compétences de l'utilisateur, le DSL le plus adapté peut être choisi pour représenter l'application. La figure 6.1 montre le DSL d'une représentation à base d'icônes.

Ainsi, une telle représentation, plus riche et plus intelligible, va permettre à l'utilisateur moyen de mieux comprendre l'application contribuant ainsi à satisfaire l'exigence d'intelligibilité [R2.1].

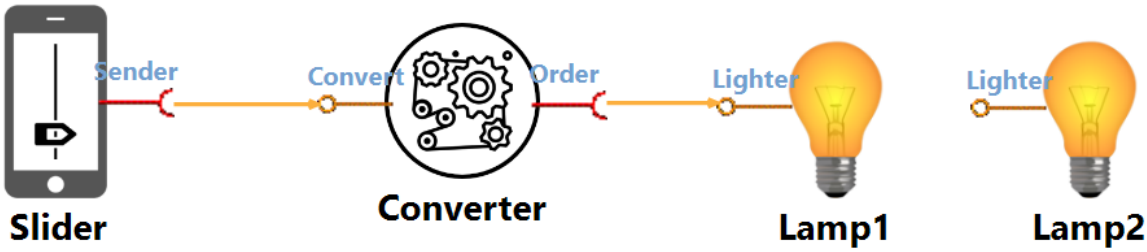


Figure 6.1 – DSL de la représentation sous forme de pictogrammes

Cependant, cette représentation reste structurelle c'est-à-dire qu'elle indique les composants qui constituent l'application et leur organisation mais pas comment s'en servir et quel service elle rend.

## 6.3 Transformation du modèle ICE en diagramme de séquence

Pour mieux être informé, l'utilisateur pourrait disposer d'informations qui explicitent la manière dont le contrôle passe d'un composant à l'autre. Pour cela, le modèle ICE est transformé en un modèle qui représente le passage du contrôle d'un composant à l'autre ; il est ensuite présenté sous la forme d'un diagramme de séquence UML [OMG, 2020b]. Ce dernier montre aussi le composant qui est directement contrôlé par l'utilisateur.

Pour être capable d'afficher le diagramme de séquence, ICE est équipé d'une extension de l'éditeur PlantUML [PlantUML, 2020] qui fournit un représentation graphique de diagrammes UML à partir d'un texte. En utilisant une transformation exogène modèle-à-texte, la première étape consiste en la transformation du modèle ICE en un modèle textuel de l'application qui est conforme avec les normes de PlantUML. L'éditeur PlantUML est ensuite appelé en lui donnant en entrée le modèle textuel généré. Ce processus va créer l'image du diagramme de séquence qui sera affichée, dans une vue externe. La figure 6.2 montre un exemple de vue.

Les méthodes de transformation de modèle facilitent donc les passages entre différents types de modèles. Ainsi, de la même manière que pour le diagramme de séquence, d'autres types de diagrammes pourraient être utilisés pour présenter à l'utilisateur des vues complémentaires. On peut envisager par exemple, la génération d'un diagramme UML de communication [OMG, 2020b]. Par l'intermédiaire d'ICE et en fonction de ses préférences, l'utilisateur pourrait choisir le type de diagramme souhaité qui serait alors généré et affiché.

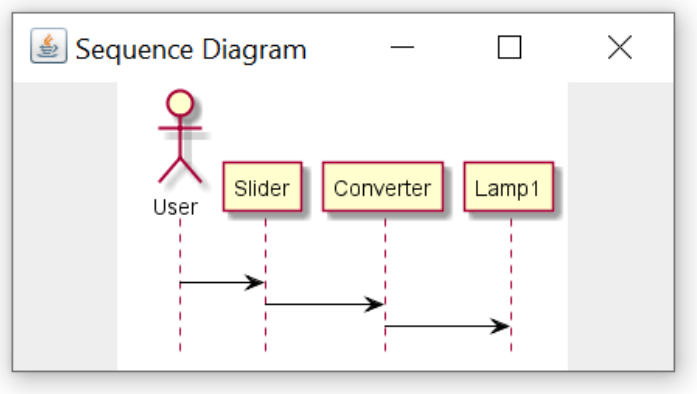


Figure 6.2 – Exemple de diagramme de séquence UML généré automatiquement

## 6.4 Transformation du modèle ICE en modèle de description

Nous devons à partir d'un modèle d'assemblage, c'est-à-dire à partir de simples descriptions de composants, de services et de liaisons, produire une sorte de mode d'emploi du service proposé. De plus, comme nous faisons l'hypothèse que l'utilisateur d'ICE est un utilisateur non spécialiste et donc non familiarisé avec la programmation, cette description doit être compréhensible pour un utilisateur moyen.

Reprenons la description de l'application proposée par ICE. D'une manière générale, les composants qui la composent sont des unités qui transforment en sorties les données passées en entrée. Les entrées ont lieu lorsqu'un service est requis avec ses paramètres ou lorsqu'une action ou un événement interne se produit. Par exemple, une entrée a lieu lorsque l'utilisateur appuie sur un bouton ou lorsqu'un capteur prend une mesure. Les sorties sont des demandes de service requis ou des actions ou événements internes, par exemple, une lampe qui s'allume ou une valeur interne du composant qui change. Ces transformations peuvent être exprimées par des règles qui décrivent comment les entrées sont transformées en sorties. Par conséquent, on peut produire une description de l'application en s'appuyant sur les règles individuelles des services présents dans l'assemblage et en les combinant pour générer des règles plus générales, portant sur toute l'application.

### 6.4.1 Description à base de règles

Nous proposons de générer des descriptions sous la forme de règles de façon à ce qu'une application émergente soit décrite par un ensemble de règles logiques. La génération se base sur trois éléments : la description des composants de l'assemblage, la description des

services, en particulier ses règles de comportement, et la description des connexions entre services. Les descriptions unitaires sont données au moment de la conception du composant. La description des connexions est donnée par OCE. La description finale est obtenue en parcourant les services de l'application et en combinant les règles des descriptions unitaires. L'ensemble des règles ainsi obtenu peut ensuite être transformé en un texte lisible pour l'utilisateur. La figure 6.3 montre une description générée automatiquement : les deux règles décrivent le comportement d'une application constituée d'un composant d'interaction ("Slider") et d'une lampe ("Lamp1").

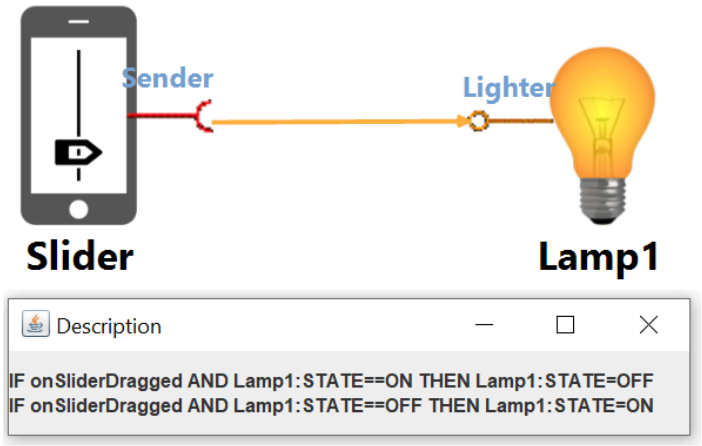


Figure 6.3 – Description à base de règles d'une application émergente

La section 6.4.2 présente le métamodèle complet et permet de voir comment le métamodèle d'assemblage (voir figure 5.1) a été étendu pour répondre aux problèmes de description. La section 6.4.3 présente le langage choisi pour décrire unitairement les composants et leur services. La section 6.4.4 présente la méthode de combinaison définie pour générer la description à base de règles. Enfin, une analyse de cette proposition ainsi que ses limites sont proposées.

### 6.4.2 Le métamodèle complet

Pour permettre une description plus riche, le métamodèle d'assemblage (Figure 5.1) a été complété avec le métamodèle de la description et le métamodèle des règles. La figure 6.4 présente l'ensemble.

Chaque composant ou service possède une description unique. Ainsi, les classes *Component* et *Service* sont composées de leur classe de description respective avec la cardinalité 1..1. Une description de composant possède 3 attributs dont 2 sont obligatoires : *Name*

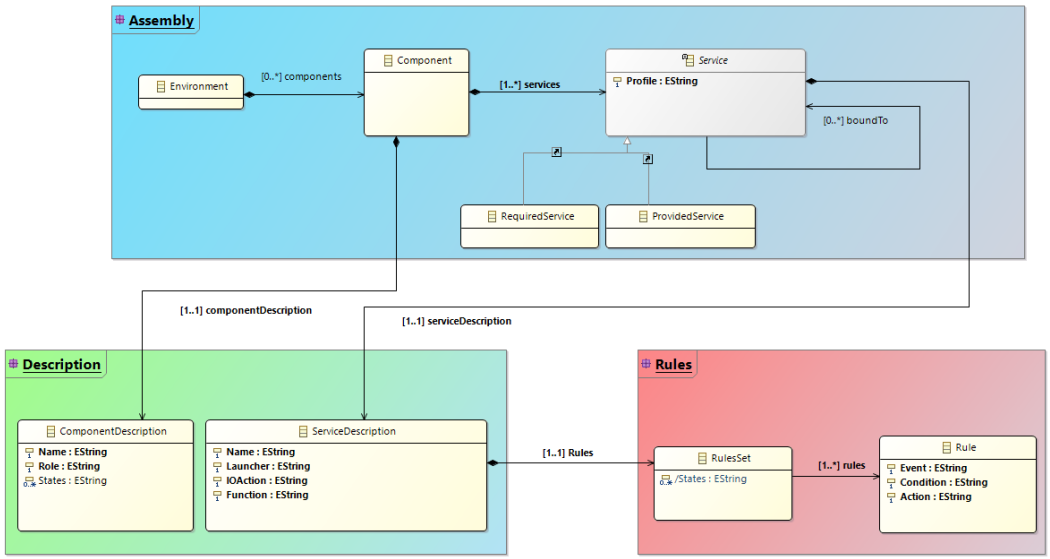


Figure 6.4 – Composition du métamodèle général

et *Role* (écrits en gras dans la classe). Le troisième attribut *States* est optionnel et son initialisation dépend du type du composant. Une description de service possède 5 attributs qui doivent être initialisés : *Name*, *Launcher*, *IOAction*, *Function* et *Rules*. Chaque service possède au moins une règle, conforme au modèle *Événement, Condition et Action* (ECA) [Berndtsson and Mellin, 2009] : une règle est conçue à partir d'un événement, d'une condition et d'une action. Ceci est exprimé par la relation de composition entre les classes *ServiceDescription* et *RulesSet*, dont la cardinalité est 1..1. La classe *RulesSet* possède aussi un attribut *States* qui est un attribut dérivé de la classe *ComponentDescription*.

Les métamodèles *Description* et *Rules* sont séparés pour des raisons d'expressivité. Notre contribution réside sur les règles de composition et non pas sur le langage de description. Le langage de description que nous avons défini n'est donc utilisé que pour montrer la faisabilité de l'approche. Il peut être changé sans affecter les règles de composition. D'ailleurs, un travail sur un langage à base d'ontologies est en cours.

La description et le rôle des attributs sont présentés dans la section suivante.

### 6.4.3 Caractéristiques du langage

Comme représenté dans le métamodèle (voir figure 6.4), la description d'un composant comporte trois attributs : *Name*, *Role* et *States*. *Name* et *Role* sont des chaînes de caractères : le premier est le nom du composant et le deuxième présente sous forme d'un texte libre le rôle

de ce dernier (par exemple, *Name*="Slider", *Role* = "Send a value"). Certains composants peuvent avoir un état interne, comme une lampe qui peut être soit allumée soit éteinte. C'est notifié avec *States* qui est l'ensemble (éventuellement vide) des états possibles d'un composant.

Un service, qu'il soit fourni ou requis, est également décrit par un ensemble d'attributs : *Name*, *Launcher*, *IOAction*, *Function* et *Rules*.

*Name* est une chaîne de caractères qui représente le nom du service.

*Launcher* est l'attribut qui définit l'événement qui déclenche le service. Sa valeur est un mot-clef prédéfini qui dépend de plusieurs cas :

1. dans le cas d'une interaction externe (provenant d'un autre composant) qui déclenche un service fourni, la valeur est *onRequired*,
2. dans le cas d'une interaction interne (provenant d'un autre service du même composant) qui déclenche le service, la valeur est *onTriggered*.
3. dans le cas d'une interaction provenant de l'utilisateur, cet attribut peut avoir plusieurs valeurs qui dépendent de la nature du composant, *p. ex.*, *onButtonPressed*, *onSliderDragged*, *onCheckBoxChecked*, ...

*IOAction* représente la manière dont le service interagit avec d'autres services. Il peut prendre les syntaxes suivantes :

1. *@OUTPUT*
2. *VAL@OUTPUT*
3. *VAL@OUTPUT :X@RETURN*
4. *VAL@RETURN*
5. *TRIGGER ServiceName*

Les trois premières syntaxes concernent l'*IOAction* d'un service requis et les deux dernières concernent l'*IOAction* d'un service fourni.

*IOAction* peut être vide pour un service fourni traitant uniquement de l'évolution de l'état du composant, sans aucune sortie (par exemple le service fourni *Lighter* qui change l'état de la lampe en ON/OFF). La figure 6.5 donne la description de ce service.

*@OUTPUT* sert à indiquer que le service émet un appel sans attribut sur l'interface de sortie d'un service requis. Par exemple, l'appel d'une fonction de type "void".

$$
\begin{aligned}
< (Name)\ Lighter,\\
(Launcher)\ onRequired,\\
(IOAction),\\
(Function)\ Turn\ ON/OFF,\\
(Rules)\ \{Launcher \wedge (STATE == OFF)\\
\Longrightarrow STATE = ON,\\
Launcher \wedge (STATE == ON)\\
\Longrightarrow STATE = OFF\} >
\end{aligned}
$$

Figure 6.5 – Description d'un service fourni pour un composant à état

*VAL@OUTPUT* correspond à un appel de fonction qui requiert une variable d'entrée. *VAL* couvre tous les types de données que le service peut traiter (comme les services ont été précédemment composés par le moteur, le problème de correspondance des types a déjà été résolu ; il est donc inutile de préciser le type).

*VAL@OUTPUT :X@RETURN* est similaire à *VAL@OUTPUT*, mais dans ce cas là, le service attend une certaine valeur en retour. Cela représente donc un appel de fonction qui retourne une donnée.

*VAL@RETURN* correspond au retour de la valeur demandée par un autre service.

La dernière possibilité pour IOAction est *TRIGGER ServiceName*. Cela définit le type de transfert de contrôle entre les services. Par exemple, dans l'application de la figure 6.1, le service *Convert* du composant *Converter* appelle le service *Order* en utilisant "*TRIGGER Order*". Pour représenter un transfert en séquence, le mot-clef TRIGGER entre les différents services est séparé par une virgule (*p. ex.*, TRIGGER Service1 , TRIGGER Service2). Dans le cas où le transfert est de type parallèle, un point-virgule est utilisé (*p. ex.*, TRIGGER Service1 ; TRIGGER Service2). Dans la section 6.4.4 nous détaillons comment ces deux types d'enchaînements sont traités et comment ils affectent le résultat final.

*Function* décrit le service sous la forme d'un texte libre (*p. ex.*, *Function* = "Turn ON/OFF"). Cet attribut est utilisé par le processus de combinaison pour générer une forme textuelle à partir de la description à base de règles (cf. section 11.3).

Comme mentionné dans la section 6.4.2, chaque service possède au moins une

règle qui sera utilisée pour combiner les descriptions unitaires ensemble et générer la description de l'application. Toutes les règles sont conformes au modèle ECA [Berndtsson and Mellin, 2009] :

1. *Événement* représente l'action qui déclenche le service. Cette action fait référence à l'attribut *Launcher* du service.
2. *Condition* représente une expression logique à satisfaire pour procéder à la partie *Action* de la règle. Par exemple "VAL@INPUT<=50" ou "STATE==OFF". *STATE* est un mot-clef qui fait référence à l'attribut *States* de la classe *RulesSet*. VAL@INPUT indique la condition sur la valeur attendue pour procéder à l'*Action*.
3. *Action* représente le résultat de l'exécution de ce service. Dans ce champ, une référence à l'attribut *IOAction* est fait.

Le choix d'utilisation du modèle ECA pour définir les règles est inspiré des travaux dans [Hamoui, 2010].

En outre, chaque partie de la règle doit être conforme à une certaine syntaxe définie par une expression régulière. La partie correspondant à *Événement* et *Condition* doit être écrite de la façon suivante :

$$Launcher[\wedge\ (STATE\ <cp>\ S)][\wedge\ (VAL@INPUT\ <cp>\ V)]\ \Longrightarrow$$

Dans cette expression $cp$ correspond à un opérateur de comparaison et peut avoir ces différentes formes : $<$, $>$, $<=$, $>=$, $==$ *et* $!=$. $S$ correspond à une valeur explicite de l'état du composant. Ses valeurs sont définies dans l'attribut *States* du composant. $V$ correspond à une valeur numérique, écrite par le fournisseur au moment de la définition de la description du service.

De la même façon, la partie *Action* doit être conforme à l'expression suivante :

$$IOAction \mid STATE = S \mid IOAction \wedge (STATE = S) \mid NOP$$

La partie *Action* possède 4 valeurs possibles : soit une référence à IOAction, soit un changement d'état, soit les deux ensemble et sinon $NOP$. Ce dernier mot-clef est utilisé si le service n'effectue aucune opération.

Pour la plupart des services, les règles sont de la forme : $Launcher \Longrightarrow IOAction$. C'est généralement le cas pour les services qui n'ont aucune condition à vérifier, autre que leur

*Launcher*, avant de déclencher leur action. Par exemple, pour qu'un *Slider* puisse émettre une valeur, il suffit que l'utilisateur bouge le curseur.

### 6.4.4 Principe de la combinaison et de la génération des règles

L'objectif est d'utiliser les descriptions unitaires des services impliqués dans l'application pour arriver à une description plus globale qui indiquerait à l'utilisateur comment l'application fonctionne. La description de l'application est donc générée à partir des règles des services et des attributs nommés par les règles (*Launcher*, *IOAction*...).

Ainsi, pour un service S1 connecté à un service S2, la ou les règles de S1 vont être combinée(s) avec celle(s) de S2 pour obtenir un ensemble de règles décrivant l'assemblage des deux services.

Pour combiner les règles, le processus cherche d'abord un mot-clé dans une règle de S1 qui soit "reliable" à un mot-clé d'une règle de S2. Par exemple, *VAL@OUTPUT* peut être relié à *onRequired* et *VAL@INPUT*. En effet, le fait que S1 fournisse une valeur en sortie (*VAL@OUTPUT*) fonctionne avec le fait que S2 attende une entrée quelconque (*onRequired*) ou une valeur (*VAL@INPUT*). De même, *TRIGGER Name* peut être relié à *onTriggered*.

Une fois deux règles R1 et R2 sélectionnées, le processus utilise un schéma de combinaison pour obtenir une seule règle R. Cette règle combinée R est à son tour combinée avec une autre règle, et ainsi de suite. Le schéma de combinaison est inspiré de la règle de coupure de la logique mathématique : à partir des règles $[\Gamma \implies A, \Delta]$ et $[\Gamma', A \implies \Delta']$, on obtient la règle $[\Gamma, \Gamma' \implies \Delta, \Delta']$.

Voici un exemple de combinaison possible :

1. R1 : $A \implies B \wedge C$
2. R2 : $C \implies D$

   R1 et R2 sont combinées en :
3. $R : A \implies B \wedge D$

Dans certains cas, il peut y avoir plusieurs règles combinées en même temps pour obtenir plusieurs règles en sortie, par exemple :

1. R1 : $A \implies B$
2. R2 : $B \wedge B' \implies C$

3. R3 : $B \wedge B'' \implies D$

   R1, R2 et R3 sont combinées en :

4. $R' : A \wedge B' \implies C$

   $R'' : A \wedge B'' \implies D$

La figure 6.6 présente le processus de combinaison pour le cas d'utilisation présenté dans la section 2.1 (page 25 - 5ème paragraphe).

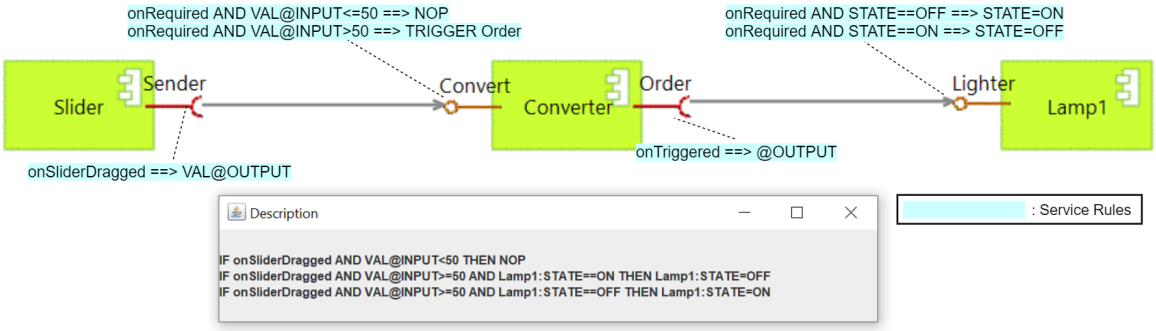


Figure 6.6 – Description à base de règles de l'application d'éclairage ambiant

Lorsque l'utilisateur définit une valeur en déplaçant le curseur du composant *Slider*, le service *Sender* est appelé avec cette valeur. *Slider* étant connecté au composant *Converter*, la valeur du slider est transmise au service fourni *Convert*. En fonction de cette valeur (plus ou moins de 50), le service *Order* est déclenché. Si *Order* est déclenché, il envoie une sortie au service *Lighter* et l'état de *Lamp1* passe de ON à OFF ou inversement. Dans la figure 6.6, la fenêtre "Description" contient les règles qui résultent de la combinaison et qui décrivent comment utiliser cette application d'éclairage ambiant.

L'exemple ci-dessus présente une topologie conforme au style architectural appelé "filtres et tubes". Cependant, notre proposition traite d'autres styles d'architecture de compositions avec des composants qui nécessitent plusieurs services, disposés en séquence, en parallèle, ou de manière conditionnelle. Ils peuvent également attendre un résultat en retour de leur demande. Ces différents exemples d'architecture sont exposés dans le chapitre 10.

### 6.4.5 Retour de données

Le retour de données concerne les cas pour lesquels certains services attendent une valeur en retour pour poursuivre leurs actions. La figure 6.7 présente un cas d'utilisation pour mieux expliquer le concept.

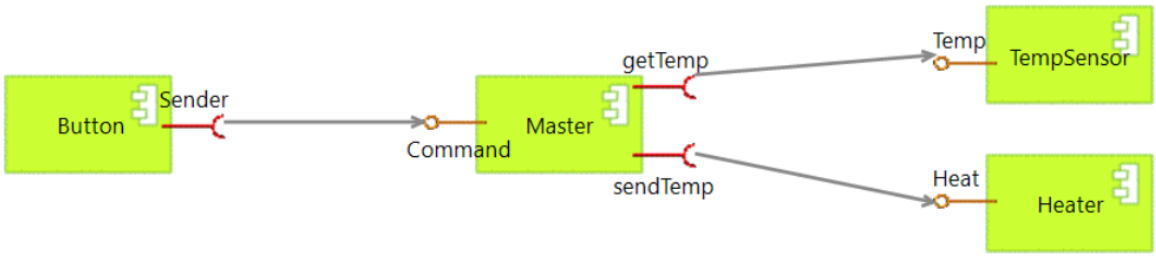


Figure 6.7 – Application pour contrôler la température ambiante

Dans ce cas, l'application vise à contrôler la température d'une chambre en allumant ou éteignant le climatiseur (représenté par le composant *Heater*) en se basant sur la valeur envoyée par le capteur. Le service *getTemp* du composant *Master* est chargé de récupérer la valeur de la température du service *Temp* du composant *TempSensor*, puis de la faire passer au service *sendTemp*. Ce dernier, à son tour, transmet la valeur au service *Heat* du composant *Heater*.

**Combinaison des règles possédant un retour de données**

Pour combiner des règles gérant le retour de données, le processus utilise une démarche similaire à celle présentée en 6.4.4. Cependant, lorsqu'il combine une règle contenant *X@RETURN* avec une règle contenant *VAL@RETURN*, il remplace *X* par la valeur qui sera retournée c'est-à-dire par *VAL* (VAL peut être remplacé par un autre attribut, *p. ex.*, TEMPERATURE@RETURN). Par exemple, la combinaison de $onButtonPressed \implies @OUTPUT :X@RETURN$ et $onRequired \implies Y@RETURN$ donne $onButtonPressed \implies Y@RETURN$.

Si le composant possède un autre service requis qui a besoin de connaître la valeur de la variable attendue en retour, cette dernière sera ensuite transmise. La figure 6.8 montre les règles des services impliqués dans l'application de la figure 6.7.

Le résultat de la combinaison des règles des services de *Button*, *Master* et *TempSensor* donnera : "$onButtonPressed \implies TEMPERATURE@OUTPUT$". Puis une combinaison de cette règle sera faite avec les règles du service du composant *Heater* en utilisant le même principe expliqué dans la section 6.4.4. Le résultat de cette combinaison donnera :

"$onButtonPressed\ AND\ TEMPERATURE@INPUT<=18 \implies Heater.STATE=ON$"

"$onButtonPressed\ AND\ TEMPERATURE@INPUT>18 \implies Heater.STATE=OFF$"

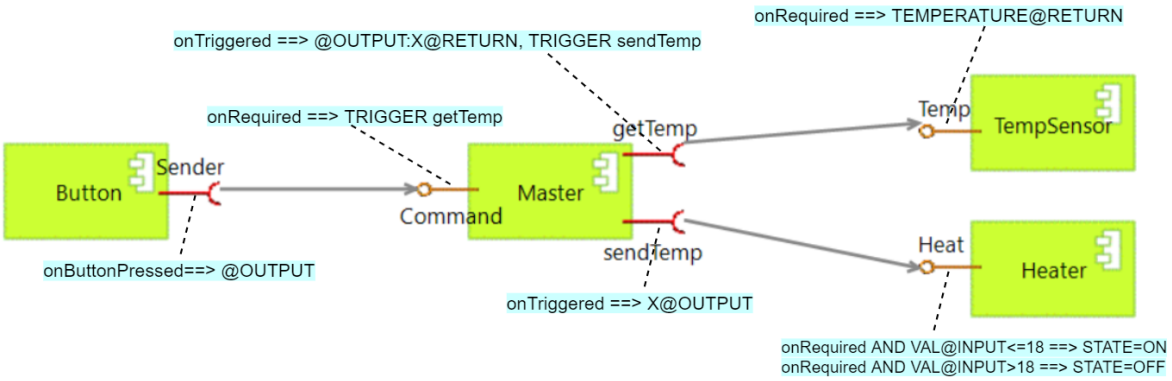


Figure 6.8 – Règles gérant le retour de données

## 6.5 Conclusion

Dans ce chapitre, plusieurs contributions permettant de mieux présenter une application aux utilisateurs ont été présentées. Grâce à l'IDM, un deuxième DSL a été réalisé. ce dernier est capable de présenter le modèle de l'application d'une façon plus compréhensible à travers un pictogramme. Plusieurs méthodes de transformations de modèle sont utilisées pour obtenir le diagramme de séquence et la description à base de règles. Cette dernière contribution est la plus originale et n'a pas été rencontrée dans l'état de l'art. Elle vise à décrire l'application en transformant le modèle de cette dernière en un ensemble de règles qui décrivent le service offert par l'application et son utilisation.

Les contributions de ce chapitre montrent comment les techniques de l'IDM permettent facilement de fournir à l'utilisateur des vues personnalisées d'une application construite par OCE. Ceci va permettre à l'utilisateur, indépendamment de son profil, de comprendre l'application qu'on lui propose et ainsi prendre une décision en connaissance de cause. L'exigence de l'intelligibilité de la présentation est donc satisfaite. De plus, grâce à la transformation de modèles, l'application peut être présentée selon différentes vues pour s'adapter à la demande de l'utilisateur.

# 7 Rôle de l'utilisateur et feedback

L'objectif de ce chapitre est de montrer comment l'utilisateur peut intervenir une fois qu'une application lui est proposée par OCE, et comment ses interventions sont prises en considération pour apprendre son besoin.

## 7.1 Interventions de l'utilisateur

Une fois l'application présentée et comprise par l'utilisateur à travers ses différentes formes, ce dernier peut décider de procéder éventuellement à la phase d'édition, puis soit à l'acceptation et au déploiement de l'application, soit à son rejet.

Pour éditer une application, l'utilisateur peut modifier ou supprimer les connexions entre les services utilisés dans l'assemblage. Il peut aussi ajouter une nouvelle connexion avec un des composants candidats présentés.

Pendant cette étape, les modifications sont contrôlées grâce aux règles définies dans le DSL (cf. section 5.2). Ces règles garantissent que l'utilisateur ne puisse faire que de modifications correctes, par exemple, qu'il ne connecte pas deux services requis. Il n'est autorisé à connecter les services que si ces derniers sont compatibles et possèdent le même profil. Ces règles se présentent sous la forme de conditions préalables et postérieures. Les conditions sont vérifiées à la volée au moment de l'édition, donc avant que les règles OCL ne soient vérifiées, pour éviter les erreurs de validation. Ceci fait partie de la réponse à l'exigence [R4.2].

De cette manière, l'utilisateur est guidé et supervisé lors de la modification des applications. Il ne peut construire que des applications correctes. Par exemple, l'application visualisée dans la figure 6.1, p. 67, présente un assemblage de trois composants et un quatrième candidat. L'utilisateur peut alors décider de remplacer *Lamp1* par *Lamp2* ou de supprimer

des composants afin d'obtenir une application consistant en un *Slider* connecté directement à une lampe. Une fois l'application modifiée, l'utilisateur peut la visualiser au moyen de son diagramme de séquence et de sa description à base de règles. Les figures 7.1 et 7.2 montrent les représentations multivues de l'application d'éclairage ambiant modifiée.

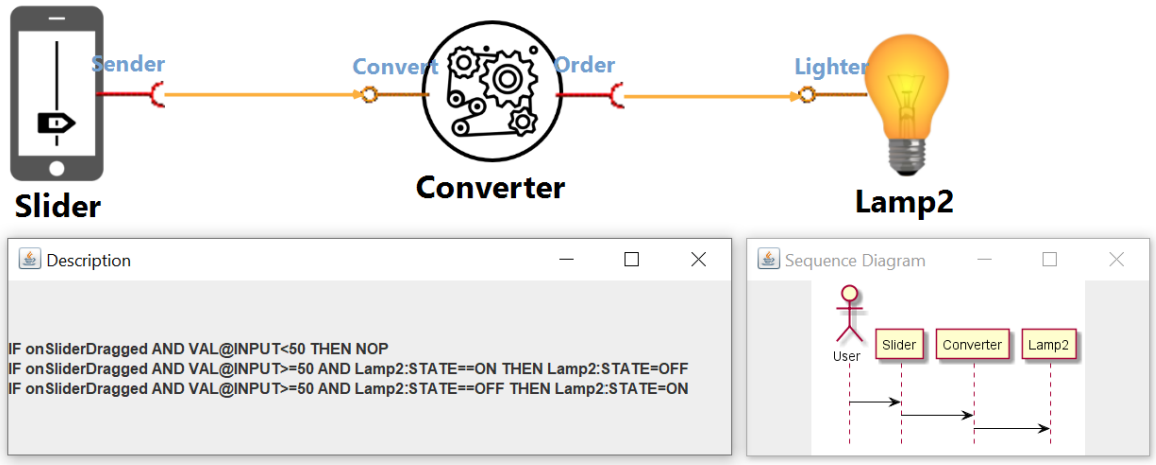


Figure 7.1 – Application d'éclairage ambiant modifiée par l'utilisateur

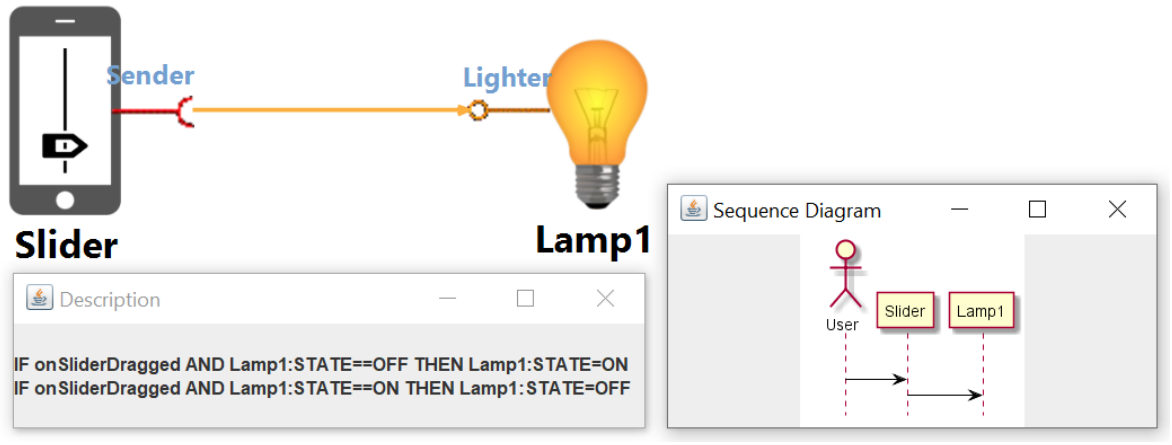


Figure 7.2 – Application d'éclairage ambiant modifiée autrement par l'utilisateur

Une fois acceptée, l'application est transformée en utilisant une transformation de modèle exogène en un script de déploiement. Ce dernier sera envoyé à une entité responsable de l'exécuter pour déployer l'application dans l'environnement.

## 7.2 Feedback

Un des problèmes est de permettre à OCE d'apprendre le besoin de l'utilisateur pour améliorer ses connaissances. Ceci est réalisé grâce à un retour sur les interventions de l'uti-

lisateur. Le feedback porte sur le fait que l'application ait été acceptée ou rejetée et sur les éventuelles modifications.

Le processus de génération de feedback se déclenche automatiquement au moment où l'utilisateur a terminé ses interventions. Une fois le feedback généré, ICE l'envoie à OCE. Ceci répond aux exigences [R5.1] et [R5.2].

L'extraction du feedback repose sur une comparaison entre deux modèles. Le premier est le modèle ICE initial (modèle de l'application émergente créé par OCE et transformé en modèle d'entrée d'ICE - cf. section 5.3) et le second est le modèle ICE final (modèle après les interventions de l'utilisateur). La comparaison s'effectue d'une part sur les composants qui ont été soit conservés dans le second modèle, soit supprimés. ICE construit une liste de composants annotés c'est-à-dire indiqués comme conservés ou supprimés. Cette liste n'étant, dans la solution actuellement implémentée, pas exploitée par le moteur OCE, elle n'est pas retournée.

D'autre part, la comparaison se fait aussi sur les connexions des services qui ont pu être modifiées (grâce à l'attribut boundTo). ICE retourne donc aussi une liste de connexions annotées. Cette liste peut contenir les éléments suivants :

1. une connexion indiquée comme inchangée.
2. une connexion indiquée comme modifiée. Une connexion est modifiée quand l'utilisateur remplace l'un des services impliqués.
3. une connexion indiquée comme supprimée.
4. une connexion indiquée comme nouvellement créée.

Les données de feedback construites par ICE sont nécessaires pour alimenter l'apprentissage par renforcement d'OCE. En effet, sans information sur l'utilité et l'utilisation des applications proposées, les connaissances d'OCE n'évolueraient pas. Rappelons que le travail sur OCE (décision et apprentissage) est effectué dans le cadre d'une autre thèse et est détaillé dans [Younes et al., 2020].

## 7.3 Conclusion

Dans ce chapitre, deux approches qui visent à assister l'utilisateur et à lui donner le privilège d'éditer ses applications ont été présentées.

La première consiste en un ensemble d'actions qui sont définies à travers les DSL et qui offrent à l'utilisateur la possibilité d'éditer son application. En outre, pour aider l'utilisateur à maintenir la pertinence des applications, les DSL contiennent des règles qui le guident à la volée pour modifier correctement l'assemblage. De cette manière, le déploiement de l'application résultante, suite aux interventions de l'utilisateur, est garanti.

La deuxième consiste à récupérer le feedback de l'utilisateur et à le transmettre à OCE. Le feedback est un élément indispensable dans le projet de composition intégrant OCE et ICE. Il permet à OCE d'apprendre le besoin de l'utilisateur. De cette manière, OCE est capable de construire des applications plus pertinentes et plus utiles pour l'utilisateur.

Ce chapitre conclut les contributions proposées dans cette thèse. La partie suivante présente l'implémentation de ces propositions pour réaliser ICE et le coupler avec OCE. Le premier chapitre présente les étapes de développement d'ICE pour intégrer les propositions : définition du métamodèle, développement des deux DSL, développement des différentes fonctions de transformation de modèle. Le chapitre qui suit présente la manière dont OCE et ICE sont couplés pour réaliser le système de composition. Enfin, les deux derniers chapitres présentent les différentes expérimentations réalisées pour valider les propositions et montrent les avantages et limites d'ICE.

*Troisième partie*

# Développement, expérimentation et analyse

# 8 Développement d'ICE

Ce chapitre présente le processus adopté pour développer ICE. Il montre comment les différents outils de l'IDM sont utilisés pour réaliser les propositions de cette thèse.

Le chapitre commence par présenter des outils. Puis, il présente les étapes du développement d'ICE. Enfin, il résume les fonctionnalités offertes par ICE.

## 8.1 Introduction

Pour développer ICE, j'ai utilisé GEMOC (cf. section 1.4.2.1) comme environnement de développement. ICE consiste en un projet EMF [Eclipse, 2020d] dont j'ai développé les différents éléments présentés dans le chapitres 5, 6 et 7. Une version stable et utilisable est disponible sur GitHub [1]. Un guide permettant de suivre les démarches pour installer ICE ainsi qu'un guide d'utilisation sont aussi disponibles.

ICE fournit des représentations multivues des applications. Il permet de présenter graphiquement les structures des applications en utilisant des DSL de type UML et à base d'icônes. De plus, le fonctionnement des applications peut être décrit à l'aide d'un diagramme de séquence UML ou à travers une description à base de règles. En plus d'être informé, l'utilisateur peut modifier, accepter ou rejeter les applications qu'on lui présente.

En outre, ICE et OCE ont été couplés et le système de composition qui en résulte est opérationnel. Associé à un environnement ambiant, il fonctionne en boucle comme le montre la figure 4.2 : les applications construites par OCE sont données en entrée à ICE, qui fournit des descriptions modifiables et compréhensibles, extrait les feedbacks des modifications éventuelles et les fournit à OCE.

Les sections suivantes présentent les étapes suivies pour réaliser ICE en commençant

1. `https://github.com/marounkoussaifi/ICE.git`

par la définition du métamodèle, puis les DSL utilisés et les différentes transformations de modèle développées pour les présentations multivues et la description à base de règles, et finalement, la gestion du feedback.

## 8.2 Étapes de développement

Dans cette section, sont présentées les différentes étapes que j'ai suivies pour développer ICE.

### 8.2.1 Définition du métamodèle

La première étape a consisté à définir le métamodèle utilisé pour le projet. En utilisant Ecore (cf. section 1.4.2.2) j'ai développé le métamodèle global d'ICE présenté dans la figure 6.4. Ecore m'a permis de définir les différentes classes avec leurs attributs, propriétés et leurs relations. En outre, j'ai utilisé OCL pour définir des règles de validation pour vérifier la compatibilité des modèles créés avec le métamodèle. Les règles OCL définies assurent les validations suivantes :

1. Deux services ne peuvent être connectés que s'ils possèdent un profil (attribut d'un service - cf. Figure 5.1 page 61) compatible.
2. Une connexion peut être créée seulement entre un service requis et un service fourni.
3. La capacité totale de connexion sur le même service ne doit pas être atteinte. Cette capacité est représentée par la cardinalité du service présent dans son attribut *Profile*.
4. Les attributs de la description des services doivent être remplis selon les mots-clefs à utiliser, par exemple, le *Launcher* d'un service fourni doit être *onRequired*.
5. La syntaxe des règles doit être valide par rapport aux expressions régulières définies pour les différentes parties.

Une fois le métamodèle défini, j'ai utilisé la procédure de génération automatique des éléments nécessaires pour utiliser ce métamodèle pour créer des éditeurs personnalisés et manipuler des modèles.

### 8.2.2 Définition de l'éditeur et des différents DSL

Après la phase de métamodélisation, j'ai crée un projet Sirius (cf. section 1.4.2.3) pour définir les DSL et un éditeur utilisable en temps réel. Pour développer les DSL, Sirius offre

un outil graphique simple à utiliser. La première étape a consisté à créer un projet de visualisation et de l'associer à un métamodèle (dans mon cas c'est le métamodèle que j'ai créé). Ce projet contient un fichier qui représente le DSL, dans lequel il faut associer chaque élément du métamodèle à sa représentation. De la même manière, il faut définir les différents menus, outils et fonctions qui permettent à l'utilisateur de modifier les modèles affichés. En outre, durant cette phase, j'ai défini les règles de contrôle pour guider l'utilisateur à modifier correctement les applications. La figure 8.1 montre l'outillage utilisé pour définir le DSL de diagramme de composants.

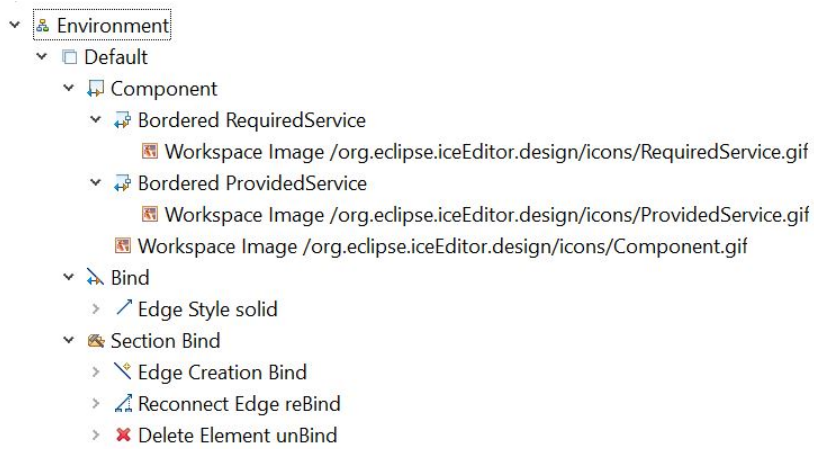


Figure 8.1 – Définition d'un DSL avec l'outil offert par Sirius

Pour réaliser l'éditeur, j'ai aussi créé un projet de modélisation [Eclipse, 2020e] que j'ai associé au métamodèle pour afficher des modèles issus de ce dernier. Par défaut, ce type de projet offre un éditeur qui doit être obligatoirement couplé avec un DSL pour afficher et manipuler les modèles.

### 8.2.3 Descriptions multivues de l'application

Les descriptions multivues suivantes ont été proposées : une représentation sous forme de pictogrammes (cf. section 6.2), une représentation sous forme d'un diagramme de séquence (cf. section 6.3) et une description à base de règles (cf. section 6.4). Les sections suivantes présentent l'implémentation de ces différentes vues.

#### 8.2.3.1 Représentation à base de pictogrammes

Pour réaliser la représentation à base de pictogrammes, j'ai configuré l'éditeur avec les deux DSL définis. Cette configuration permet d'afficher simultanément un modèle XML en

entrée de l'éditeur sous forme d'un diagramme de composants et d'un diagramme à base d'icônes.

La figure 8.2 montre un exemple d'une partie du modèle ICE.

```
<?xml version="1.0" encoding="UTF-8"?>
<iCE_Editor:Environment xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:iCE_Editor="http://www.eclipse.org/sirius/sample/ice_editor">
<component>
    <service xsi:type="iCE_Editor:RequiredService" Profile="..." boudTo="..."\>
</component>
...
```

Figure 8.2 – Exemple du modèle XML d'ICE

Selon son profil, l'utilisateur peut ainsi choisir la représentation qu'il souhaite. À noter qu'une modification réalisée sur une représentation est automatiquement reportée sur l'autre. La figure 8.3 montre une même application représentée simultanément avec les deux DSL.

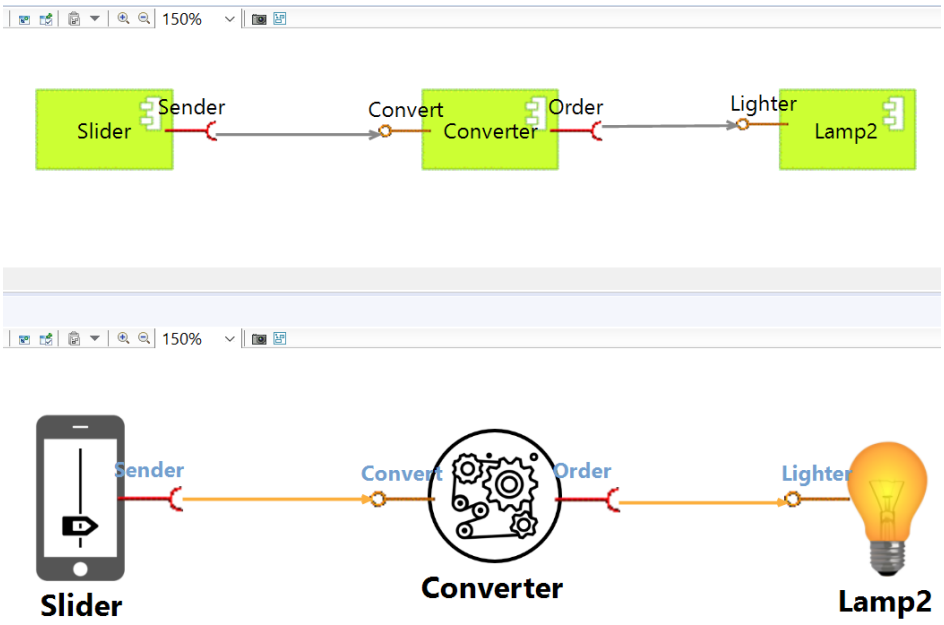


Figure 8.3 – Représentations simultanées d'une même application

L'outil Sirius a été utilisé pour réaliser le DSL pour visualiser l'application sous forme de pictogrammes, . Une méthode dynamique est implémentée qui permet au DSL d'afficher les icônes des composants dans le pictogramme. Pour réaliser cette méthode, une fonction Java a été développée dans la classe "Services" que le projet Sirius offre. Cette classe est utilisée pour résoudre des contraintes complexes ou pour traiter un processus dynamique

qui visualise des modifications à la volée. La figure 8.4 montre l'implémentation de cette fonction.

```
public String getImageFromComponent(EObject self) {
    Bundle bdl = Platform.getBundle(Activator.PLUGIN_ID);
    Component comp = (Component) self;
    String imageName = comp.getName().replaceAll(" ", "");
    Path imagePath = new Path("/icons/"+imageName);
    URL imageURL = FileLocator.find(bdl, imagePath, null);
    if(imageURL != null) {
        return "/averageuser.ice.design/"+imagePath.toString();
    }
    return "/averageuser.ice.design/icons/Converter.png";
}
```

Figure 8.4 – Fonction d'affichage dynamique des icônes du pictogramme

Cette fonction cherche, dans le répertoire d'icônes, le nom de l'icône du composant sélectionné et retourne son chemin d'accès pour l'attribuer dynamiquement au composant. Enfin, cette fonction est appelée avec une requête de type "Service" dans l'attribut de personnalisation de la disposition du noeud composant.

Alternativement, il existe une méthode générique pour obtenir l'affichage du pictogramme. Cette dernière consiste à attribuer au noeud (représentant le composant) une disposition conditionnelle pour chaque possibilité, *i.e.* définir toutes les icônes possibles qu'un composant peut avoir. La figure 8.5 montre un exemple.

Cependant cette méthode est plus difficile à gérer dans le cas où il existe plusieurs représentations possible d'un composant.

#### 8.2.3.2 Représentation sous forme d'un diagramme de séquence

Comme expliqué dans la section 6.3, j'ai utilisé une transformation de modèle exogène pour transformer le modèle de l'application en un diagramme de séquence UML. Pour réaliser cette tâche, j'ai d'abord étendu mon éditeur avec une extension de PlantUML, puis j'ai implémenté une méthode hybride de transformation de modèle qui utilise les langages Java et AQL [Eclipse, 2020b] simultanément. AQL est un langage utilisé pour naviguer dans un modèle EMF et permet de faire des requêtes sur ce dernier.

La première étape consiste à utiliser cette méthode hybride pour générer le modèle textuel du diagramme de séquence, compatible avec le langage PlantUML. L'algorithme de génération du modèle textuel est donné par l'algorithme 8.1.

**Algorithme 8.1 :** Transformation du modèle ICE en diagramme de séquence

```
Result : Modèle textuel du diagramme de séquence
initialiser un fichier contenant la configuration initiale du fichier textuel du
  diagramme de séquence : file;
foreach component ∈ environment do
    vérifier si c'est un composant dont son service requiert une interaction humaine;
end
if True then
    ajouter à file la commande qui relie l'acteur à ce composant;
    while tous les composants présents dans l'application ne sont pas sélectionnés do
        foreach service ∈ component do
            if détection de parallélisme dans les règles du service then
                ajouter à file l'attribut pour exprimer le début du parallélisme;
                while tous les composants qui forment le processus parallèle ne sont pas
                  utilisés do
                    trouver le composant connecté à ce service;
                    ajouter à file le texte nécessaire pour exprimer cette relation;
                end
                ajouter à file l'attribut pour exprimer la fin du parallélisme;
            else
                trouver le composant connecté à ce service;
                ajouter à file la commande nécessaire pour exprimer la relation entre
                  ces deux composants;
            end
        end
    end
else
    faire le même processus en choisissant un composant aléatoire pour commencer
      avec;
end
sauvegarder file ;
```

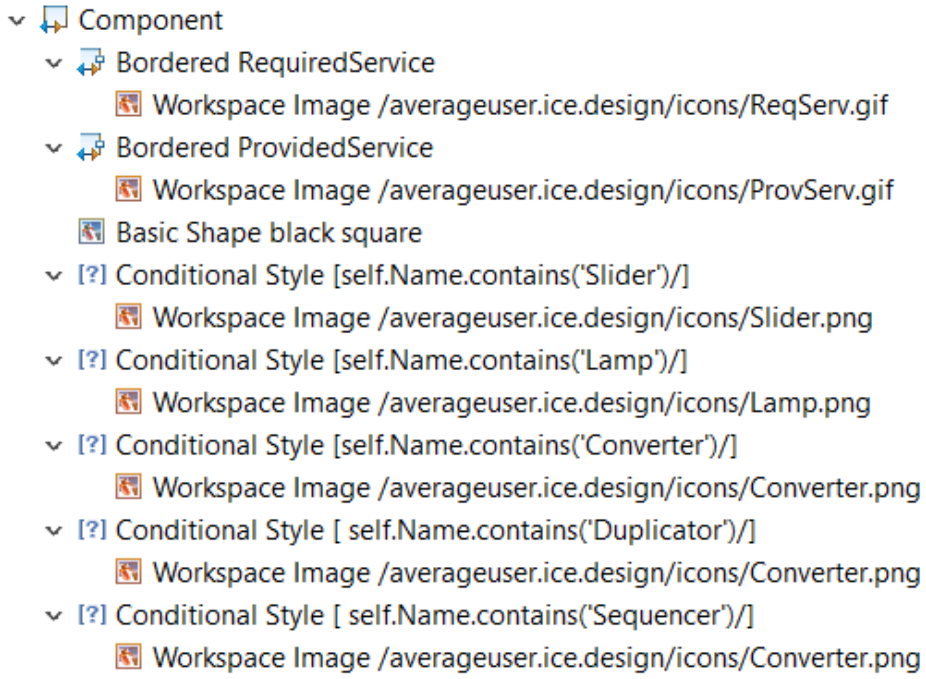


Figure 8.5 – Méthode alternative pour l'affichage des icônes

Actuellement, le processus de génération de diagramme de séquence fonctionne correctement pour la majorité des types d'architectures des applications (cf. section 11.1).

Ensuite, la deuxième étape consiste à utiliser la requête PlantUML ci-dessous pour générer le diagramme de séquence :

$$java\ -jar\ plantuml.jar\ Description.txt\ -o\ output$$

*Description.txt* représente le nom du fichier contenant le modèle textuel du diagramme de séquence résultant de l'algorithme 8.1 (voir l'image centrale de la figure 8.7), et *output* représente le répertoire où l'image du diagramme de séquence sera stockée après sa génération. L'image créée aura le même nom que son modèle textuel, mais avec une extension différente (*.png* dans mon cas).

Enfin, la fonction Java de la figure 8.6 affiche l'image résultante dans une vue pop-up :

La figure 8.7 présente le processus pour obtenir le diagramme de séquence.

La partie gauche de la figure représente le modèle de l'application visualisé par l'utilisateur à travers l'éditeur. La partie du milieu montre le modèle textuel (qui se trouve entre les deux mots-clefs "*@startuml*" et "*@enduml*") qui représente le diagramme de séquence PlantUML. Ce modèle a trois parties :

```java
private void showSequenceDiag() {
    try {
        frame = new JFrame("Sequence Diagram");
        ImageIcon icon = new ImageIcon("output\\Description.png");
        JLabel label = new JLabel(icon);
        frame.add(label);
        frame.pack();
        frame.setVisible(true);
        frame.toFront();
        icon.getImage().flush();
    }catch(Exception e) {
        e.printStackTrace();
    }
}
```

Figure 8.6 – Fonction d'affichage du diagramme de séquence

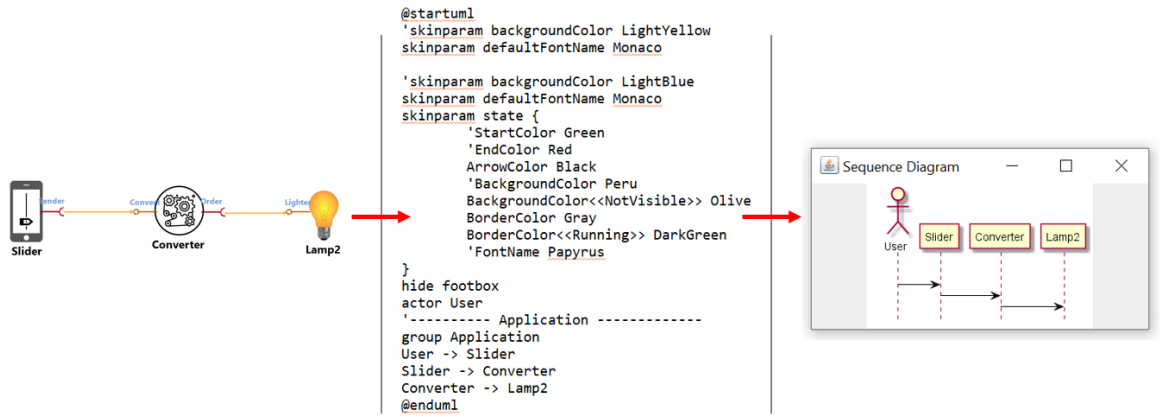


Figure 8.7 – Processus de génération du diagramme de séquence PlantUML

1. Définition du style du diagramme et personnalisation de la disposition. Cette partie commence de "*@startuml*" jusqu'à le mot-clef "*hidefootbox*"
2. Définition des acteurs principaux du système par le mot-clef "*actor*".
3. Définition du diagramme de séquence en se servant des composants et des services qui composent l'application (ces éléments se trouvent entre "*groupApplication*" jusqu'à "*@enduml*").

La partie à droite montre l'image du diagramme affichée à l'utilisateur après exécution de la commande qui transforme le modèle textuel en modèle graphique.

#### 8.2.3.3 Description à base de règles

Pour générer la description de l'application, j'ai également utilisé une transformation hybride en utilisant Java et AQL qui transforme les règles des services de l'application en une ou plusieurs règles qui décrivent cette dernière.

La première étape consiste à trouver un composant avec lequel l'utilisateur peut faire son interaction. Grâce à AQL on parcourt le modèle affiché pour trouver un composant dont son service possède un *Launcher* avec une valeur qui indique une interaction utilisateur, *p. ex.*, *onButtonPressed*. Puis la ou les règles du service trouvé avec l'AQL sont combinées avec celles du service connecté à ce dernier (trouvé dans l'attribut *boundTo* de sa description) en suivant le processus expliqué dans la section 6.4.4. Une fois générées, ces règles vont être à leur tour combinées avec celles du prochain service. Ce processus est répété jusqu'à l'utilisation de toutes les règles des services connectés. Le résultat final est une liste de règles. L'algorithme 8.2 décrit ce processus.

### 8.2.4 Génération du feedback

Comme expliqué dans la section 7.2, la génération du feedback se fait grâce à une comparaison entre le modèle originalement proposé par OCE et le modèle qui résulte de l'intervention de l'utilisateur. La comparaison de modèles est basé sur un développement hybride comme pour les processus de la section précédente, pour lequel j'ai utilisé le langage Java couplé avec le langage AQL.

Plus précisément, AQL a été utilisé pour naviguer dans les modèles et Java pour le codage de la fonction de comparaison. La fonction de comparaison est utilisée pour construire le feedback sous une forme adéquate pour le moteur OCE. Ce processus de transmission du retour de l'utilisateur au moteur OCE est présenté dans le chapitre suivant sur l'intégration d'ICE et d'OCE.

## 8.3 Conclusion

Le développement d'ICE répond aux exigences de départ. Il est opérationnel et son code se trouve sur GitHub [2]. Le prototype intègre les contributions décrites dans les chapitres précédents.

ICE est capable de présenter l'application émergente sous plusieurs représentations structurelles qui sont compréhensibles par des utilisateurs de profils différents (expert ou utilisateur moyen). De plus, ICE offre à l'utilisateur les descriptions sémantiques sur la fonction de l'application sous forme de diagrammes de séquence et de descriptions à base de

2. `https://github.com/marounkoussaifi/ICE.git`

**Algorithme 8.2 :** Transformation du modèle ICE en description à base de règles

```
1  initialiser une variable textuelle vide : description;
2  foreach component ∈ environment do
3  |  vérifier si c'est un composant dont son service requiert une interaction humaine;
4  end
5  if True then
6  |  Faire une liste de tous les composants de l'application en commençant par le
   |   composant dont son service requiert une interaction humaine;
7  |  while tous les composants présents dans liste ne sont pas sélectionnés do
8  |  |  //Lorsqu'un composant est sélectionner dans l'algorithme qui suit, il est
   |  |   automatiquement supprimer de liste
9  |  |  foreach service ∈ component do
10 |  |  |  combiner les règles de service avec les règles de service trouvé dans la
   |  |  |   propriété boundTo;
11 |  |  |  Faire un ou des séries qui représentent l'ensemble de services qui forment
   |  |  |   une chaîne entre le service actuellement sélectionné dans boundTo
   |  |  |   jusqu'à arrivé au service du dernier composant;
12 |  |  |  //Il faut faire attention aux règles pour détecter les architectures de
   |  |  |   l'application
13 |  |  |  //Dans le cas de parallélisme, nous obtenons deux séries de services
14 |  |  |  //Dans le cas de Filtres et Tubes, nous obtenons une seule série de
   |  |  |   services
15 |  |  |  while tous les services de ou des séries ne sont pas utilisés do
16 |  |  |  |  récupérer les règles du prochain service et les combiner avec les règles
   |  |  |  |   dernièrement obtenues;
17 |  |  |  end
18 |  |  end
19 |  end
20 else
21 |  faire le même processus en choisissant un composant aléatoire pour commencer
   |   avec;
22 end
23 Attribuer les règles obtenues à la variable description;
24 Présenter le résultat (description) à l'utilisateur;
```

règles.

ICE permet à l'utilisateur de configurer son application s'il le souhaite, puis de l'accepter ou la rejeter.

Actuellement, ICE est dans sa première version stable. Il existe encore quelques points à traiter ou à améliorer. Ces derniers sont présentées dans la section 11.1 du chapitre 10.

Nous allons voir maintenant comment ce prototype s'intègre avec OCE pour réaliser le système de composition.

# 9 Intégration avec OCE

Pour fonctionner, ICE a besoin d'informations provenant d'OCE et inversement. Ce chapitre présente notre solution pour faire interopérer ICE et OCE. Cette solution a été développée en collaboration avec Walid Younes, en charge du composant OCE.

Après une brève introduction, les exigences et les caractéristiques techniques de la solution sont présentées. Puis les modes de fonctionnement entre ICE et OCE sont détaillés. Enfin, le chapitre fait une synthèse de la solution et présente les perspectives d'amélioration de cette dernière, la phase de déploiement de l'application est notamment explorée.

## 9.1 Introduction

Pour intégrer les deux parties du système de composition (OCE et ICE), nous avons développé une extension appelée M[OI]CE. Cette extension, développée en se basant sur l'IDM, est chargée de gérer les échanges d'informations entre OCE et ICE.

Étant une extension commune, M[OI]CE est divisée en deux parties. La première, implémentée dans le moteur, est chargée de transformer le modèle OCE en un modèle compatible avec ICE et de le transmettre à ce dernier pour être présenté à l'utilisateur sous la forme souhaitée. La deuxième, implémentée dans ICE, permet de calculer le feedback de l'utilisateur et de l'envoyer à OCE pour mettre à jour ses connaissances. En outre, M[OI]CE doit envoyer le script de déploiement au système, une fois que l'utilisateur a validé son application.

La section suivante présente les exigences que nous avons définies et les différents modes de fonctionnement de M[OI]CE.

## 9.2 M[OI]CE : Exigences et principe

### 9.2.1 Exigences

L'extension M[OI]CE est située entre OCE et ICE. Elle est constitué d'une partie chargée des communications vers ICE et d'une autre partie chargée des communications vers OCE. Les exigences relatives à chaque partie sont exposées.

#### 9.2.1.1 Exigences ICE

Le rôle d'ICE est de présenter à l'utilisateur les applications émergentes construites par OCE. Or OCE utilise un modèle textuel non compatible avec l'entrée requise par ICE. ICE doit recevoir une description de l'application qui satisfait les exigences suivantes :

1. la description doit indiquer les composants et les services qui constituent l'application, ainsi que les connections entre services,
2. cette description doit être conforme au métamodèle d'assemblage utilisé par ICE (voir figure 5.1),
3. les composants compatibles et leurs services candidats doivent aussi être transmis.

#### 9.2.1.2 Exigences OCE

Le rôle d'OCE est de construire des applications émergentes intéressantes pour son utilisateur. Pour cela, il doit recevoir un retour de son utilisateur sur l'intérêt de l'application proposée. OCE doit recevoir une description de l'application qui satisfait les exigences suivantes :

1. la description doit être une liste de connexions de services,
2. chaque connexion doit être annotée comme acceptée, refusée, ajoutée (suite à la modification de l'application) ou remplacée (suite à la modification de l'application).

### 9.2.2 Principe de la solution

La solution que nous proposons pour intégrer ICE avec OCE et qui répond aux exigences d'ICE et d'OCE est représentée dans la figure 9.1.

Les différentes étapes de fonctionnement du système de composition (incluant l'interaction avec M[OI]CE) sont les suivantes :

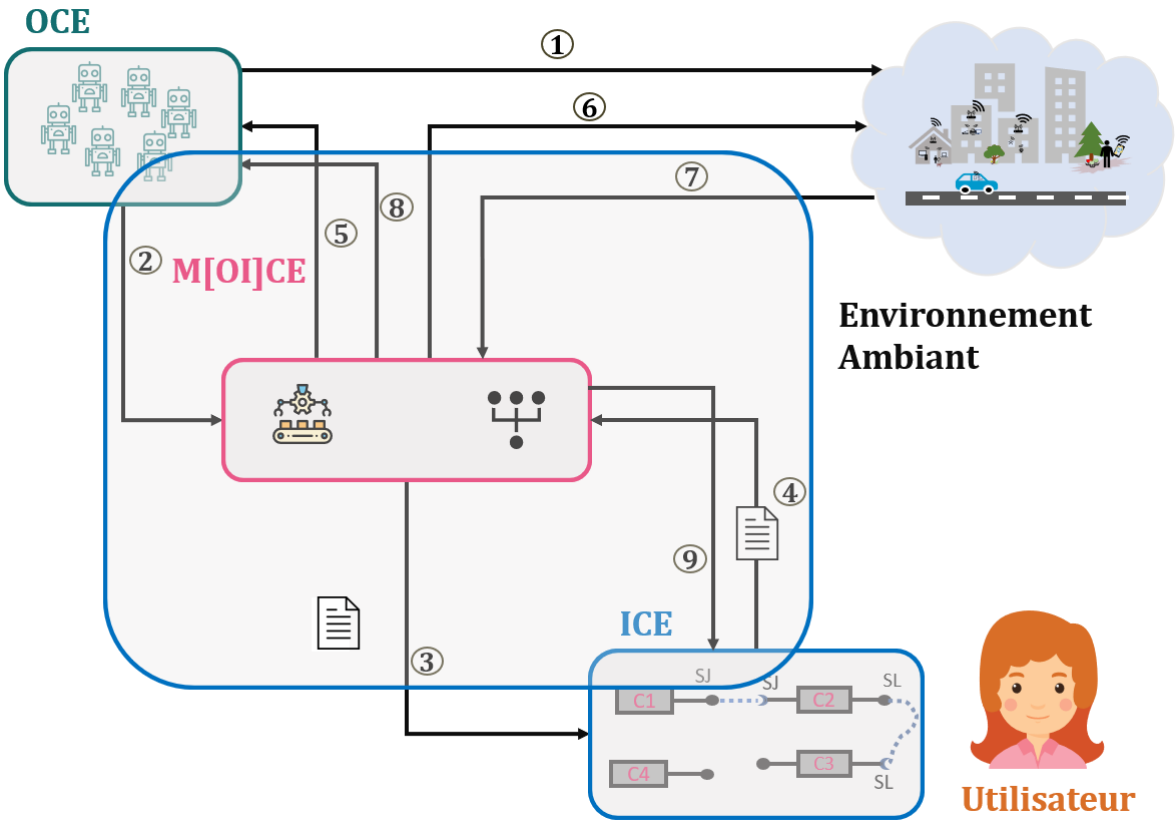


Figure 9.1 – Architecture du système, incluant M[OI]CE

1. OCE sonde l'environnement. Il associe un agent à chaque service détecté dans l'environnement. Les agents coopèrent entre eux et proposent des connexions.
2. À la fin du cycle moteur, l'ensemble des connexions (qui constitue le modèle de configuration) et la liste des services disponibles sont envoyés à M[OI]CE.
3. M[OI]CE les transforme en un modèle d'entrée (fichier xml) conforme au métamodèle utilisé par ICE et le transmet à ICE.
4. Via l'éditeur d'ICE, l'utilisateur modifie éventuellement des connexions, puis accepte ou refuse l'application présentée. Après que l'utilisateur a fini ses interventions, la description de l'application est transmise à M[OI]CE.
5. À la réception du nouveau modèle de configuration de l'utilisateur, M[OI]CE compare le modèle d'entrée (fichier xml envoyé à ICE) et cette nouvelle configuration (fichier xml envoyé en retour). Il envoie ce retour à OCE.
6. En même temps, M[OI]CE déclenche le déploiement de l'application dans l'environnement ambiant.

7. S'il y a d'éventuelles erreurs lors du déploiement effectif, M[OI]CE les transmet à OCE (étape 8) et à l'utilisateur via ICE (étape 9).

## 9.3 M[OI]CE : Implémentation

L'architecture interne de M[OI]CE est présentée via un diagramme de composant dans la figure 9.2.

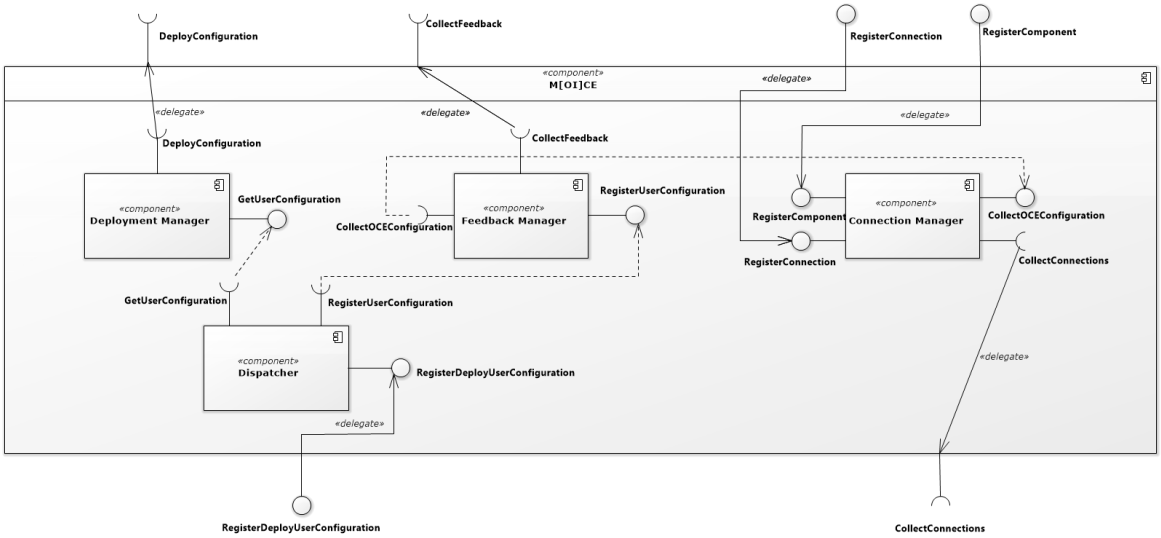


Figure 9.2 – Architecture interne de M[OI]CE

M[OI]CE est constitué de 4 composants.

Le composant *Connection Manager* :

— établit et maintient la connexion entre OCE et ICE,

— envoie à ICE le modèle d'application proposé par OCE,

— récupère d'ICE les données du feedback.

Le composant *Feedback Manager* :

— reçoit les données de feedback du *Connection Manager*,

— analyse les données pour la mise à jours de la connaissance d'OCE,

— envoie les données d'apprentissage à OCE.

Le composant *Deployment Manager* génère le script nécessaire pour déployer l'application après sa validation et envoie sa configuration au *Dispatcher* pour la sauvegarder pour des utilisations ultérieures.

Enfin, le composant *Dispatcher* a pour rôle de garder la trace de tous les données qui concernent le feedback et le déploiement. Ces données sont sauvegardées en prévision de futurs travaux sur les sources possibles de feedback.

Les figures qui suivent présentent le diagramme de classes de chacun de ces composants. Le diagramme de classes du composant *Connection Manager* est représenté dans la figure 9.3.

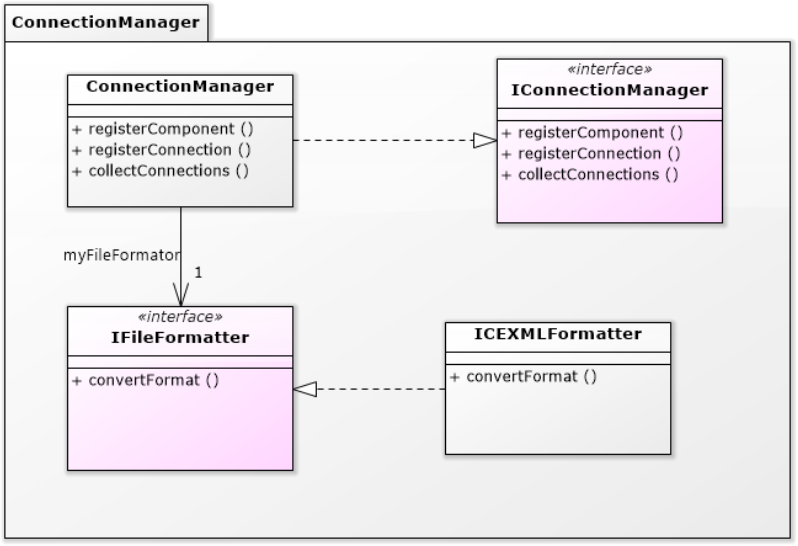


Figure 9.3 – Diagramme de classes du *Connection Manager*

Le paquetage est composé de deux classes et de deux interfaces. La classe *ConnectionManager* récupère le modèle de l'application proposée par OCE en utilisant les fonctionnalités fournies par l'interface *IConnectionManager* et le transforme en un modèle compatible avec ICE (fichier XML) grâce à la fonction *convertFormat*.

Le diagramme de classes du composant *FeedbackManager* est représenté dans la figure 9.4.

Le paquetage est composé de deux classes. *FeedbackManager* génère le feedback et le transmet à OCE pour mettre à jours sa connaissance. Avant de le transmettre, les données du feedback sont écrites dans un fichier compatible avec OCE. Ceci est géré par la classe *OCEFileFormatter*. Actuellement, l'implémentation de ce diagramme est réalisée en totalité.

Le diagramme de classes du composant *DeploymentManager* est représenté dans la figure 9.5.

Le paquetage est composé de deux classes. *DeploymentManager* transforme le modèle validé par l'utilisateur en un script de déploiement dans l'environnement réel. Ensuite le

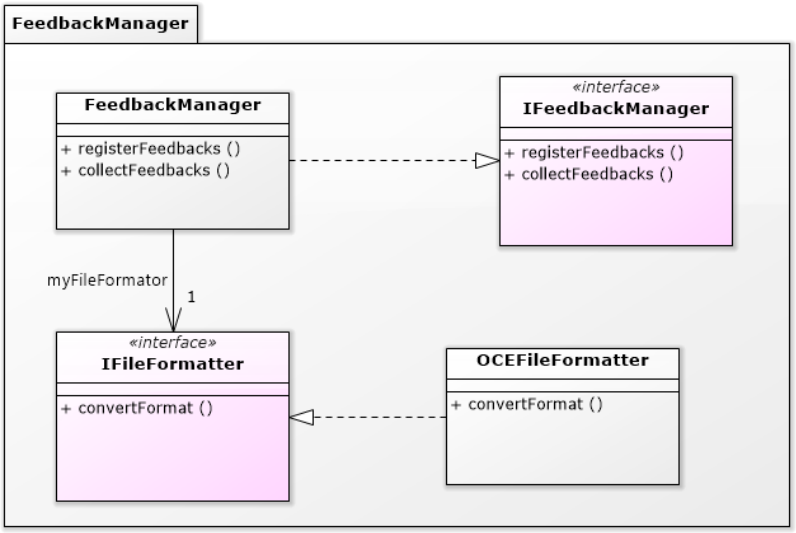


Figure 9.4 – Diagramme de classes du *FeedbackManager*

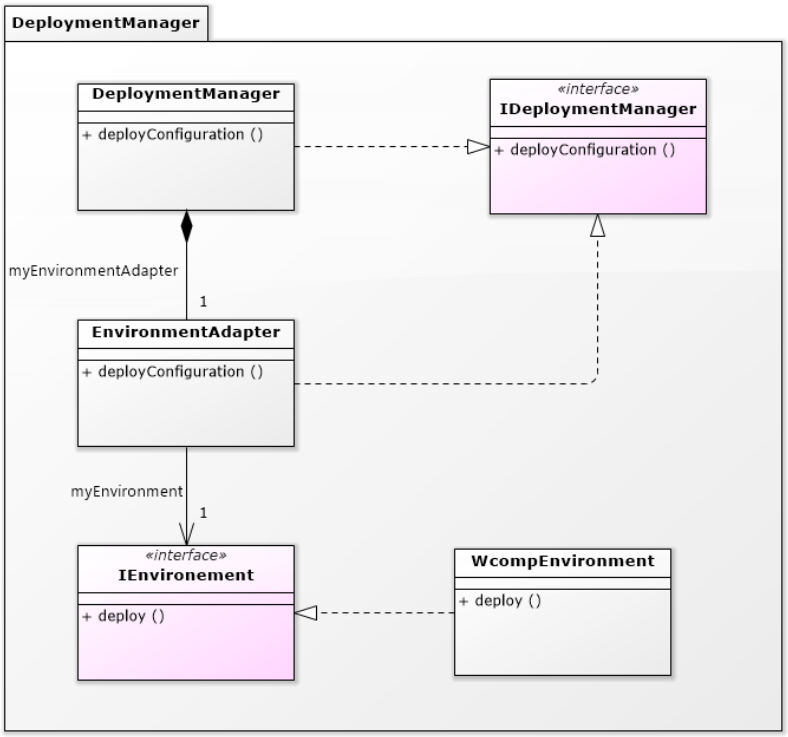


Figure 9.5 – Diagramme de classes du *DeploymentManager*

script est transmis à un adaptateur spécifique qui a comme rôle d'analyser le script et de l'exécuter pour réaliser l'application. Ceci est représenté par la classe *EnvironmentAdapter*. Une fois l'application émergente déployée, l'utilisateur sera notifié et pourra ainsi l'utiliser.

À ce jour, seuls les deux premiers composants ont été développés et sont en produc-

tion. Le composant *DeploymentManager* sera implémenté ultérieurement. Le diagramme présenté ici, décrit l'architecture technique dans le cas d'un déploiement impliquant WComp [Rainbow, 2006] qui est un environnement de simulation et d'exécution pour les applications d'intelligence ambiante.

### 9.3.1 Transformation du modèle OCE en modèle ICE

C'est la classe ICEXMLFormatter (voir figure 9.3) qui est en charge de transformer le modèle OCE en modèle ICE (les détails de cette transformation ont été présentés dans la section 5.3). Pour cela, la méthode *convertFormat()* suit l'algorithme 9.1 :

**Algorithme 9.1 :** Transformation du modèle OCE en modèle ICE

```
1  initialiser une liste contenant les composants proposés par OCE : listComponents;
2  initialiser un fichier xml vide : file;
3  foreach component ∈ listComponents do
4      écrire dans file la représentation xml du composant;
5      initialiser listServices = component.getProvidedServices();
6      foreach service ∈ listServices do
7          if service.getBoundTo.size() > 0 then
8              écrire dans file la représentation xml du service en remplissant tous ses
                 attributs;
9              // les attributs à remplir sont type, Name, Launcher, IOaction, Function,
                 Profile, boundTo et Rules
10         else
11             écrire dans file la représentation xml du service en remplissant tous ses
                 attributs sauf boundTo;
12         end
13         // il faut vérifier le type de service pour remplir l'attribut type avec la valeur
             requis ou fourni
14     end
15 end
16 sauvegarder file;
17 envoyer file à ICE;
```

### 9.3.2 Calcul du feedback

La classe *FeedbackManager* (voir figure 9.4) génère le feedback en utilisant la méthode *collectFeedbacks()*. Cette dernière effectue une comparaison de modèle entre le modèle d'OCE et celui d'ICE comme indiqué dans l'algorithme 9.2. Cette méthode est capable de déduire les modifications faites par l'utilisateur en comparant le modèle de l'application originalement

proposé par OCE et le modèle après validation ou rejet de l'utilisateur.

**Algorithme 9.2 :** Génération du feedback

```
1  initialiser une liste contenant les composants proposés par OCE :
     componentsOCEConfiguration;
2  initialiser une liste contenant les composants après la validation de l'utilisateur :
     componentsICEUserConfiguration;
3  foreach component ∈ componentsOCEConfiguration do
4      Chercher si component se trouve dans componentsICEUserConfiguration;
5      if component ∈ componentsICEUserConfiguration then
6          initialiser une liste serviceListOCEComponent contenant les services de
             component ∈ componentsOCEConfiguration;
7          initialiser une liste serviceListICEComponent contenant les services de
             component ∈ componentsICEUserConfiguration;
8          foreach service ∈ serviceListOCEComponent do
9              comparer service avec sa version dans serviceListICEComponent pour
                 déduire l'état de sa connexion : acceptée, supprimée, créée ou modifiée;
10         end
11     else
12         //dans ce cas, le composant a été supprimé par l'utilisateur
13         //pour l'instant, ce cas n'est pas traité par OCE
14     end
15 end
16 envoyer le feedback à OCE;
```

M[OI]CE transmet le feedback ainsi généré à OCE.

La comparaison de modèle qui se fait dans cette fonction ici est basée sur la méthode et les technologies expliquées dans 8.2.4.

## 9.4 Conclusion

Ce chapitre a présenté la solution pour intégrer OCE et ICE en un système de composition opérationnel. La solution a consisté à développer un composant logiciel intermédiaire M[OI]CE composé de deux parties. La première a été développée dans OCE pour transformer le modèle proposé par OCE en une version compatible pour ICE, c'est-à-dire en un modèle visualisable dans l'éditeur graphique. La deuxième a été développée dans ICE pour générer un feedback et l'envoyer à OCE pour qu'il puisse apprendre des préférences de l'utilisateur.

Actuellement, grâce à M[OI]CE, le système de composition fonctionne correctement en

suivant le principe de la solution proposée dans 9.2.2 : le système peut présenter à son utilisateur une application émergente, sous une forme que ce dernier peut comprendre, et en retour, le système est capable de prendre en compte les préférences de son utilisateur pour de nouvelles propositions d'application. Cependant, deux composants restent à développer pour arriver à la version pleinement opérationnelle de M[OI]CE.

Un développement supplémentaire porte sur la phase de déploiement de l'application après sa validation par l'utilisateur. Actuellement, grâce à l'utilisation d'Acceleo, ICE est capable de transformer le modèle en un code Java qui simule le déploiement. Cependant, le phase de déploiement consiste à réaliser l'assemblage proposé dans un environnement réel pour permettre à l'utilisateur de profiter du service offert. Ceci pourra être réalisé avec une transformation du modèle ICE en un vrai script de déploiement qui sera transmis par M[OI]CE à l'unité responsable de son exécution.

Le deuxième développement consiste à superviser le déploiement pour détecter les exceptions causées par l'environnement. En effet, il est possible qu'entre le moment où OCE construit le modèle de l'application et le moment où l'application est déployée, des composants impliqués dans l'assemblage aient disparu. Le déploiement est alors compromis. M[OI]CE devrait pouvoir détecter un éventuel problème de déploiement et avertir l'utilisateur que l'application n'est pas disponible. Alors, OCE démarra un autre cycle moteur.

Le chapitre suivant présente les expérimentations que nous avons réalisées pour vérifier l'intégration de M[OI]CE et son fonctionnement. Il présente aussi d'autres expérimentations réalisées pour valider les propositions présentées dans les chapitres précédents.

# 10 Expérimentation

Ce chapitre présente les différentes expérimentations réalisées pour valider ICE. Elles ont été réalisées afin de valider l'ensemble des contributions proposées dans cette thèse.

Ce chapitre commence par présenter les résultats validant l'intégration d'OCE et d'ICE. Ensuite, il décrit les résultats des expérimentations faites pour valider les représentations et les descriptions qu'ICE offre à l'utilisateur.

## 10.1 Introduction

Une première version d'ICE a été mise en œuvre. Elle s'appuie sur GEMOC studio, Eclipse Modeling Framework, EcoreTools, Sirius et Acceleo (comme mentionné dans le chapitre 8).

Actuellement, ICE fournit une représentation multivue d'une application. Il permet de présenter graphiquement la structure de l'application en utilisant deux DSL : le premier permet une représentation sous forme d'un diagramme de composants UML et le deuxième permet une représentation à base d'icônes. De plus, le fonctionnement des applications peut être décrit à l'aide de diagrammes de séquence UML et dans un langage à base de règles. Avec ICE, en plus d'être informé, l'utilisateur peut modifier, accepter ou rejeter ces applications.

Le système de composition dans sa globalité résulte du couplage d'OCE et d'ICE. Il est pleinement opérationnel. Associé à un environnement ambiant, OCE et ICE fonctionnent en boucle, conformément à ce qui a été illustré dans la figure 4.2. Quand les applications construites à la volée par OCE lui sont transmises, ICE fournit des représentations modifiables et des descriptions intelligibles à l'utilisateur, extrait ses feedback, et les transmet en retour à OCE. Le couplage OCE - ICE a été essentiel pour les expérimentations, d'ICE bien

entendu, mais aussi pour la solution d'apprentissage d'OCE qui a pu être expérimentée et validée par ailleurs [Younes et al., 2020].

Pour valider ICE (et son intégration avec OCE), différents cas d'utilisation ont été mis en oeuvre. Ils illustrent l'émergence d'applications d'architectures de différents types :

— Filtres et Tubes : l'application est composée de composants connectés sous la forme d'un enchaînement linéaire. Les appels commencent du premier service (appartenant au premier composant) et vont jusqu'au dernier (appartenant au dernier composant).

— Parallèle : l'application offre deux ou plusieurs services qui sont rendus de manière simultanée. Dans cette architecture, il existe un composant qui possède deux (ou plusieurs) services requis et un service fourni. Le service fourni de ce composant active simultanément les services requis.

— Séquentielle : cette architecture est semblable à la précédente. C'est l'ordre des appels qui diffère. En effet, les services requis d'un composant sont activés l'un après l'autre (en séquence) par le service fourni.

— Retour de données : on considère ici le cas d'un service requis qui récupère, en retour, un résultat au service fourni avec qu'il est connecté. Les données récupérées peuvent être traitées par le composant en interne ou transmises en paramètre à un autre appel.

L'objectif de ces expérimentations était de valider la faisabilité des contributions et plus précisément par l'évaluation des résultats obtenus sur les points suivantes :

1. La transmission à ICE d'un modèle transformé à chaque fois qu'OCE termine un cycle moteur. Cette tâche est gérée par M[OI]CE.
2. La présentation structurelle de l'application en utilisant les différents DSL définis.
3. La possibilité pour l'utilisateur d'éditer et de modifier les représentations structurelles.
4. Les restrictions et les orientations offertes à l'utilisateur pour assurer une bonne édition des applications durant son intervention
5. La validation ou de le rejet de l'application.
6. Les transformations de modèle pour obtenir les diagrammes de séquences UML et les descriptions basées sur des règles.
7. L'extraction du feedback de l'utilisateur et de sa transmission à OCE.

La section 10.2, montre au travers d'un exemple, le couplage entre OCE et ICE. Elle valide le point 1. La section 10.3 présente les interventions possibles par l'utilisateur. Elle montre qu'ICE offre un ensemble de commandes suffisant pour permettre à l'utilisateur d'exprimer ses choix. Elle valide les points 2, 3, 4, 5 et 7. La section 10.4 déroule des cas d'utilisation amenant à différents types d'architectures d'applications. Pour chacun de ces types, ICE est capable de générer les descriptions adéquates. Elle valide le point 6. Finalement, ce chapitre se termine par une analyse des résultats obtenus et par une mise en évidence des points forts et des limites de notre solution.

## 10.2 Transition d'OCE vers ICE

Plusieurs tests ont été réalisés pour valider la transition entre OCE et ICE réalisée par l'implémentation de M[OI]CE. Cette transmission des données d'OCE vers ICE est illustrée par l'exemple ci-dessous : il s'agit d'un cas d'utilisation où OCE propose d'assembler un bouton existant sur la tablette de l'utilisateur avec une lampe dans sa chambre.

La figure 10.1 représente l'interface graphique d'OCE réalisée par Walid Younes pour ses besoins d'expérimentation [Younes et al., 2020] : cette interface permet de suivre les actions du moteur, en particulier de visualiser le modèle d'assemblage qui émerge. Celui-ci est représenté par le graphe affiché à la droite de la figure où l'on voit deux agents avec une connexion entre leurs services.

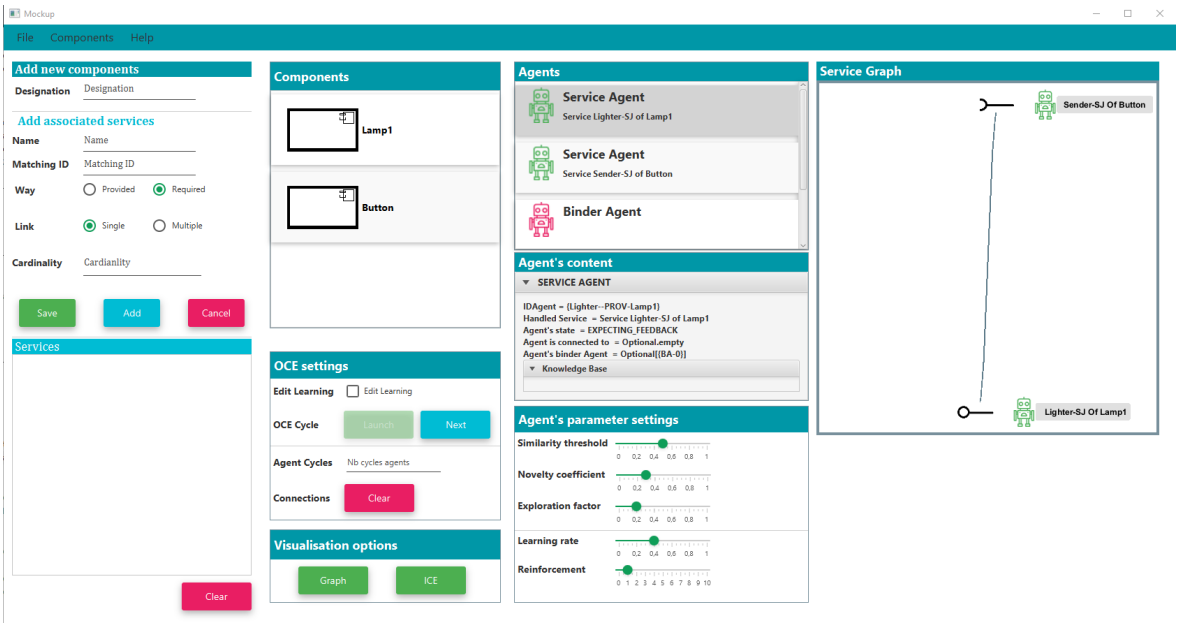


Figure 10.1 – Interface d'OCE affichant les informations sur la composition en cours

Lorsqu'OCE a achevé son cycle moteur, le modèle est transmis à M[OI]CE pour être transformé en un modèle compatible avec ICE. Le modèle transformé est ensuite transmis à ICE pour être affiché dans deux représentations différentes (correspondant aux deux DSL) : l'une à base de composants UML, l'autre à base d'icônes comme le montre la figure 10.2.

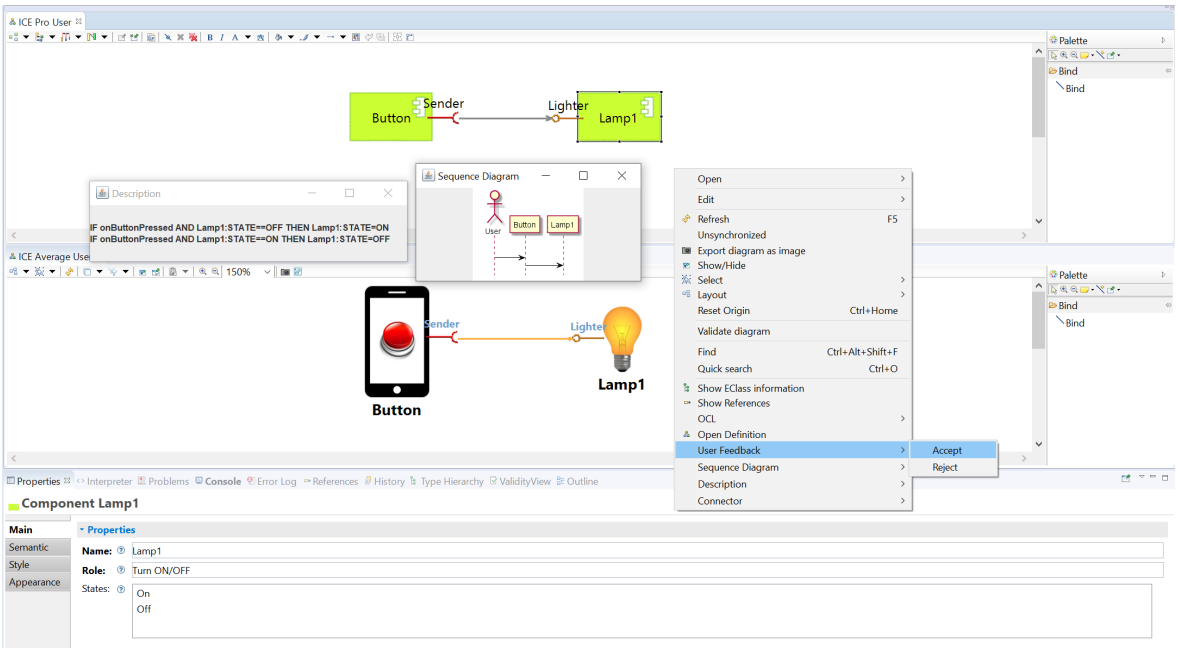


Figure 10.2 – Interface ICE affichant l'application proposée par OCE

M[OI]CE est donc opérationnel, l'ensemble OCE-ICE forme ainsi un système de composition complet. Ce dernier a pu ainsi être utilisé dans plusieurs projets comme :

1. Dans les expérimentations et les validations de la thèse de Walid Younes (surtout pour l'apprentissage à travers le feedback généré par ICE)

2. Le projet d'assistance intelligente et proactive en environnement professionnel (AILP) qui consiste à développer un assistant virtuel intelligent permettant à un opérateur en milieu industriel d'interagir avec son environnement par la création d'applications composée d'objets connectés et de services disponibles. Il s'agit d'un projet de recherche en cours soutenu par la région Occitanie et co-financé par les fonds Feder/FSE Midi-Pyrénées et Garonne 2014-2020. Il réunit des partenaires académiques et professionnels. OCE et ICE sont des composants essentiels de la solution logicielle actuellement en cours de développement.

3. Des expérimentations avec des composants logiciels réels comme celle d'un utilisateur dans une voiture électrique qui bénéficie d'un service émergent pour le guider vers la

station de recharge de batterie la plus proche. Une démonstration peut être vue ici : `https://www.irit.fr/~Sylvie.Trouilhet/demo/DelcourtM1Demo.mp4`

## 10.3 Interventions de l'utilisateur

L'objectif principal d'ICE est de permettre à l'utilisateur d'intervenir dans son environnement ambiant, modifier ses applications et éventuellement les déployer ou les rejeter.

### 10.3.1 Assistance à l'utilisateur

Comme expliqué dans le chapitre 7, l'utilisateur, grâce aux DSL, aux règles de contrôle, et aux outils d'édition, peut modifier ses applications tout en étant assisté par ICE, ce qui assure la création d'applications fonctionnelles.

En complément, ICE dispose d'un menu dédié au contrôle de l'utilisateur (voir figure 10.3). À travers ce menu, l'utilisateur peut explicitement demander la description à base de règles de son application (item "*Description*") et son diagramme de séquence (item "*Sequence Diagram*"). Ces options supplémentaires de présentation permettent à l'utilisateur de mieux appréhender l'application proposée.

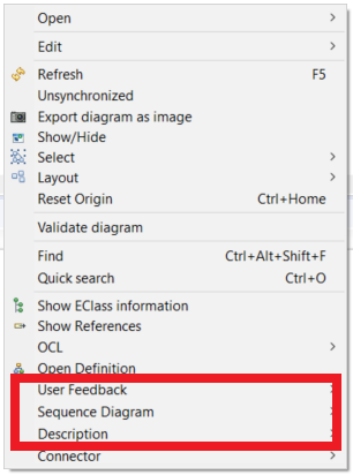


Figure 10.3 – Options du menu dédiées à l'utilisateur

De plus, durant l'étape d'édition, l'utilisateur peut sélectionner un élément (composant ou service) pour avoir une description plus détaillée de ce dernier : pour un composant, son nom, son rôle et son état, pour un service, son nom, son profile, sa fonction et son comportement (*Profile*, *IOAction*, *Launcher* et *Rules*). Ce zoom sur un composant ou sur un service peut être utile pour l'utilisateur par exemple s'il envisage de remplacer un service de l'ap-

plication par un service candidat. Les figures 10.4 et 10.5 montrent les fenêtres d'affichage des descriptions d'un composant et d'un service.

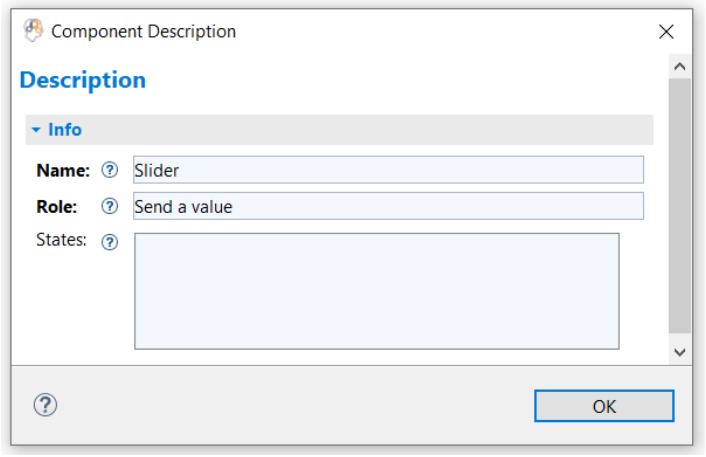


Figure 10.4 – Fenêtre de description d'un composant

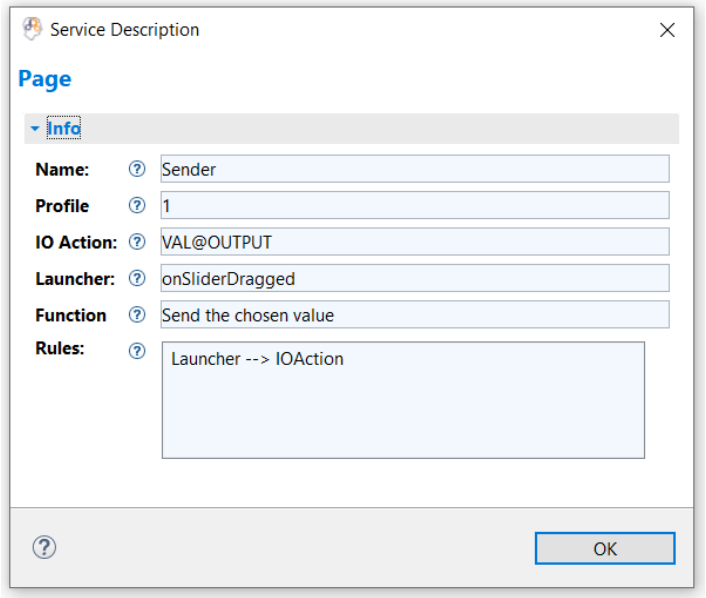


Figure 10.5 – Fenêtre de description d'un service

Après avoir visualisé et éventuellement modifié l'application, l'utilisateur utilise l'item "*User Feedbak*" du menu pour accepter ou rejeter l'application émergente. Cette dernière intervention de l'utilisateur déclenche l'algorithme de calcul du feedback opéré par M[OI]CE.

### 10.3.2 Prise en compte du feedback

Dans cette section, sans entrer dans les détails du mécanisme d'apprentissage du moteur OCE qui n'est pas dans le périmètre de cette thèse (la solution d'apprentissage et les détails

de l'attribution des valeurs est présenté dans [Younes et al., 2020]), nous voulons seulement montrer qu'OCE prend bien en compte les feedback extraits des actions de l'utilisateur dans ICE.

Dans l'exemple présenté dans la figure 10.2, l'utilisateur décide d'accepter l'application. Pour cela, il choisit "Accept" de "*User Feedbak*". Ce choix déclenche automatiquement le processus de calcul de feedback géré par M[OI]CE pour transférer les données d'apprentissage à OCE pour faire la mise à jour de sa connaissance. La figure 10.6 montre comment la connaissance d'un agent a été modifiée.

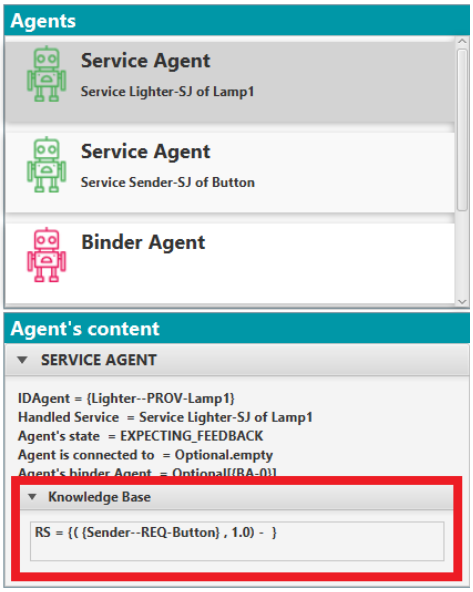


Figure 10.6 – Interface d'OCE montrant la mise à jour sa connaissance après l'acceptation de l'utilisateur

Comme le feedback vient d'une acceptation sans modification, OCE a attribué la connaissance de l'agent responsable du service *Lighter* de *Lamp1* par une valeur de 1.0 (visible dans le rectangle rouge). Cette valeur traduit le fait que l'utilisateur a accepté l'utilisation de ce service.

En outre, la génération du feedback couvre aussi les situations suivantes :

— L'utilisateur a d'abord modifié son application en remplaçant la *Lamp1* par la *Lamp2* puis il l'a acceptée : la génération du feedback détecte ce changement et le transmet à OCE.

  La figure 10.7 montre la prise en compte de ce feedback. À la réception du feedback, OCE met à jour la connaissance du service du composant *Button*. Pour ce composant,

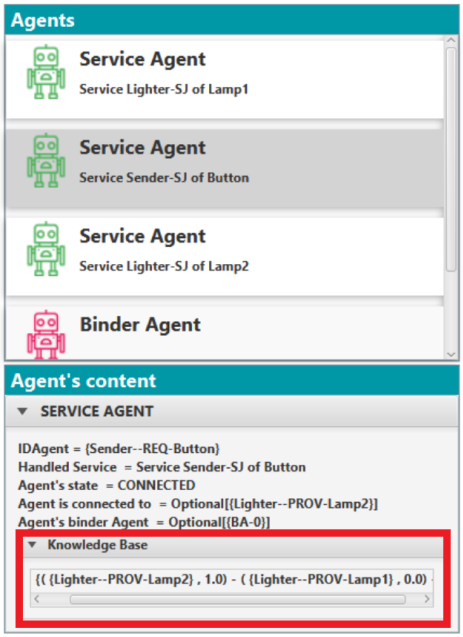


Figure 10.7 – Interface d'OCE montrant la mise à jour de sa connaissance après la modification de l'application et l'acceptation par l'utilisateur

sa connaissance montre qu'il est préférable de se connecter avec la *Lamp2* au lieu de la *Lamp1* en attribuant les valeurs 1.0 et 0.0 (visibles dans le cadre rouge).

— L'utilisateur a rejeté l'application, ce qu'indique le feedback renvoyé à OCE.

La figure 10.8 montre la prise en compte du rejet. Comme la connexion entre *Button* et *Lamp1* est rejetée, OCE ainsi attribut la valeur 1.0 à *Lamp2* et 0.0 à *Lamp1*. En conséquence, dans un prochain cycle moteur, OCE privilegiera la connexion entre *Button* et *Lamp2*.

## 10.4 Description de plusieurs formes d'architecture

Il est important de vérifier qu'ICE est capable de manipuler différents schémas d'assemblage. Aussi, nous avons défini des cas d'utilisation amenant à l'émergence d'applications d'architectures différentes.

### 10.4.1 Architecture de type Filtres et Tubes

La figure 10.9 montre un cas d'utilisation simple où l'application créée vise à allumer une lampe en déplaçant un curseur sur un smartphone. Cette dernière est composée de deux composants ayant chacun un service. Voici les règles de chaque service (les mêmes

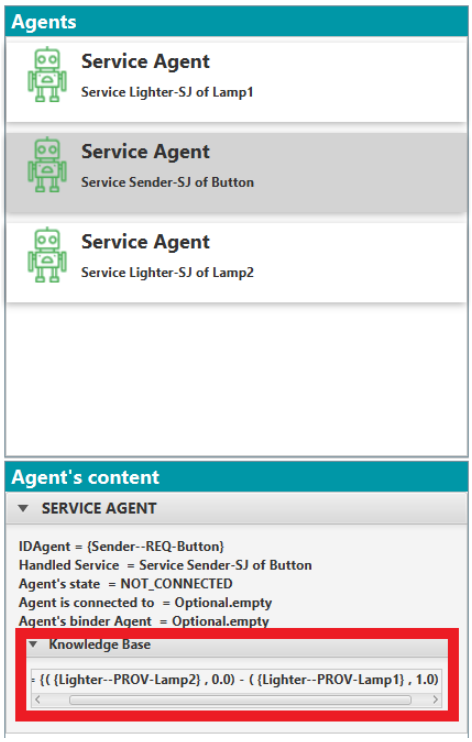


Figure 10.8 – Interface d'OCE montrant la mise à jour de sa connaissance après un rejet total de l'application

règles seront utilisées dans tous les cas d'utilisation où ces services apparaîtront) :

**Service Sender.** onSliderDragged $\Longrightarrow$ VAL@OUTPUT

**Service Lighter.** onRequired & STATE==OFF $\Longrightarrow$ STATE==ON

onRequired & STATE==ON $\Longrightarrow$ STATE==OFF

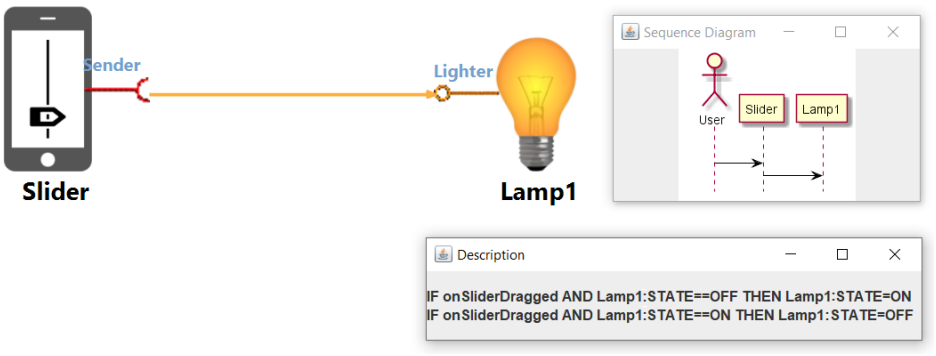


Figure 10.9 – Application d'éclairage ambiant - Architecture Filtres et Tubes simple

On constate qu'ICE est capable de présenter l'application, de créer son diagramme de séquence et de générer sa description. On peut remarquer que, grâce à la description, la lampe est directement contrôlée par le *Slider* sans aucune condition particulière autre que

son état.

La figure 10.10 montre un deuxième cas d'utilisation dans le quel un composant intermédiaire est introduit. Il s'agit du composant *Converter* ayant deux services dont voici les règles :

**Service Convert.** onRequired & VAL@INPUT<50 $\Longrightarrow$ NOP
onRequired & VAL@INPUT>=50 $\Longrightarrow$ TRIGGER Order

**Service Order.** onTriggered $\Longrightarrow$ @OUTPUT

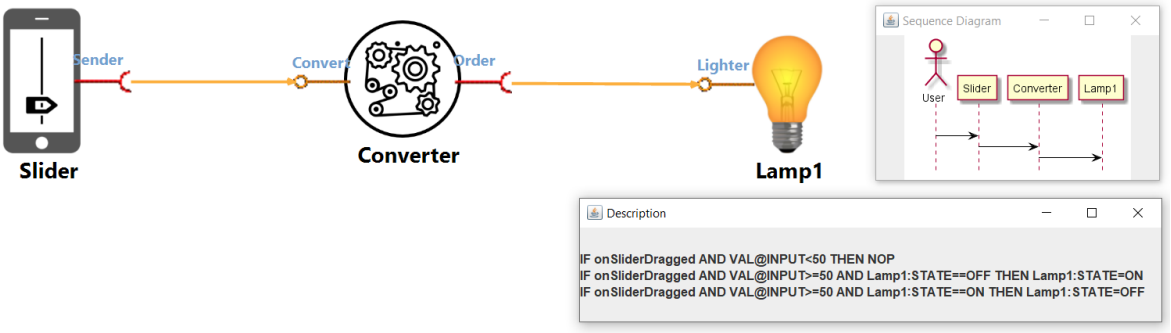


Figure 10.10 – Application d'éclairage ambiant - Architecture Filtres et Tubes à plusieurs composants

Ici, *Lamp1* change d'état sous condition, ces dernières étant exprimées par *Converter*.

Le processus de génération de règles continue de bien fonctionner lorsque plusieurs composants sont utilisés : les règles sont combinées (mot-clé AND) et montrent clairement les conditions d'utilisation de l'application. Il en est de même pour la transformation du modèle en diagramme de séquence.

### 10.4.2 Architecture parallèle

La figure 10.11 montre un cas d'utilisation où plusieurs composants sont activés en parallèle. Dans cette application, l'utilisateur contrôle simultanément deux lampes. Le contrôle des composants *Lamp1* et *Lamp2* est possible grâce au composant *Duplicator* qui duplique le signal reçu sur son service fourni *Duplicate* et l'envoie simultanément aux services requis *Order1* et *Order2*. Ces deux services possèdent les mêmes règles que le service *Order* du cas d'utilisation précédent. Voici les règles du service *Duplicate* :

**Service Duplicate.** onRequired $\Longrightarrow$ TRIGGER Order1 ; TRIGGER Order2

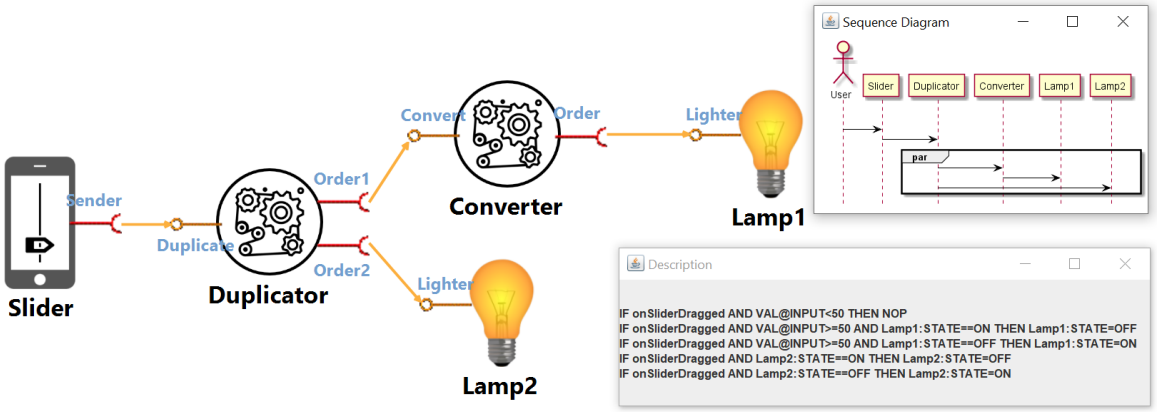


Figure 10.11 – Application d'éclairage ambiant - Architecture parallèle

La description à base de règles indique que les deux lampes sont contrôlées, en même temps, en utilisant le *Slider* : cinq règles de même niveau commençant par onSliderDragged.

Pour la génération du diagramme de séquence, le système est capable de détecter le parallélisme en inspectant les règles. Lors de la transformation du modèle, le diagramme de séquence créé, contient l'attribut "par" (voir le diagramme de séquence affiché) qui indique que les appels s'effectuent en parallèle.

### 10.4.3 Architecture non linéaire et séquentielle

La figure 10.12 montre un cas d'utilisation proche de celui présenté dans la section précédente, mais dans lequel le composant *Sequencer* remplace le *Duplicator*. Lorsque le service fourni du *Sequencer* est appelé, les services *Order1* et *Order2* sont appelés l'un après l'autre, c'est-à-dire que lorsque *Order1* termine son action, *Order2* commence la sienne. L'objectif de cette application est donc d'allumer deux lampes, l'une après l'autre. Voici les règles du service *Arrange* :

**Service Arrange.** onRequired $\Longrightarrow$ TRIGGER Order1 , TRIGGER Order2

La description à base de règles indique que les deux lampes sont contrôlées séquentiellement (le système est donc capable de détecter et de décrire des actions séquentielles) : elle contient trois règles (en parallèle) puis, séparées par le mot-clef *NEXT*, deux règles (en parallèle). Le diagramme de séquence montre l'enchaînement séquentiel des actions.

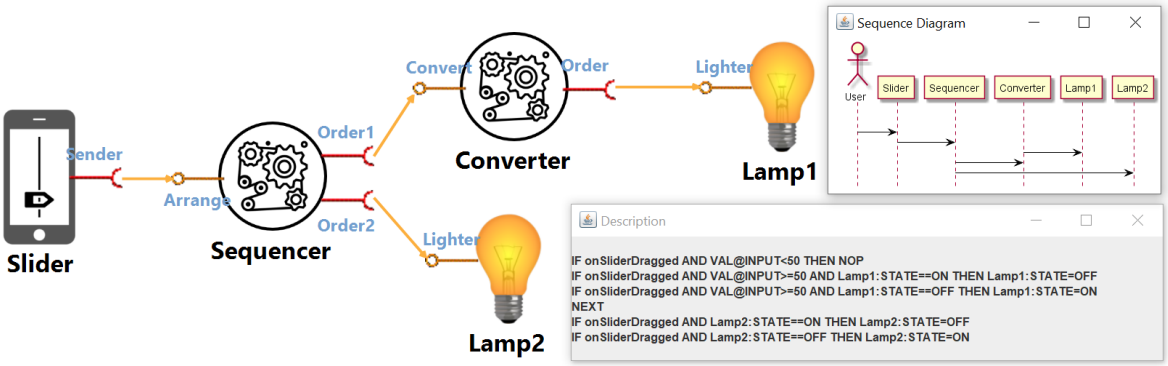


Figure 10.12 – Application d'éclairage ambiant - Architecture séquentielle

### 10.4.4 Retour de données

La figure 10.13 montre un cas d'utilisation où un service (*getValue*) attend une donnée en retour.

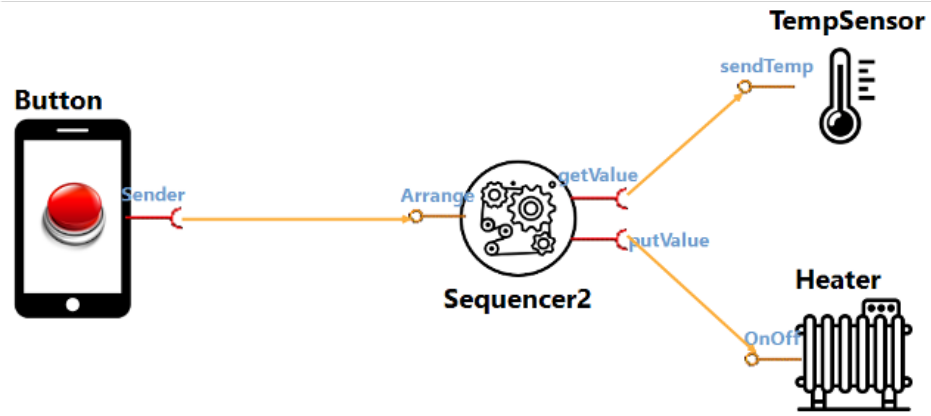


Figure 10.13 – Application de réglage de la température ambiante

L'application vise à contrôler la température ambiante en allumant ou éteignant le chauffage. Ce contrôle est régit par la valeur de la température envoyée par le composant *TempSensor*. La température fournie *sendTemp* est demandée par le service *getValue* qui, à sa réception, va la transmettre au service *putValue* pour l'envoyer au service fourni *OnOFF* du composant *Heater*.

Voici les règles des six services :

**Service Sender.** onButtonPressed $\Longrightarrow$ @OUTPUT

**Service Arrange.** onRequired $\Longrightarrow$ TRIGGER getTemp

**Service getValue.** onTriggered $\Longrightarrow$ @OUPUT :X@RETURN , TRIGGER Command

**Service sendTemp.** onRequired $\Longrightarrow$ TEMPERATURE@RETURN

**Service putValue.** onTriggered $\Longrightarrow$ X@OUTPUT

**Service OnOff.** onRequired & VAL@INPUT<=18 $\Longrightarrow$ STATE=ON
onRequired & VAL@INPUT>18 $\Longrightarrow$ STATE=OFF

La figure 10.14 montre le diagramme de séquence de l'application. Le diagramme exprime le retour d'information par la flèche qui a comme origine *TempSensor* et qui part en direction de *Sequencer2*.

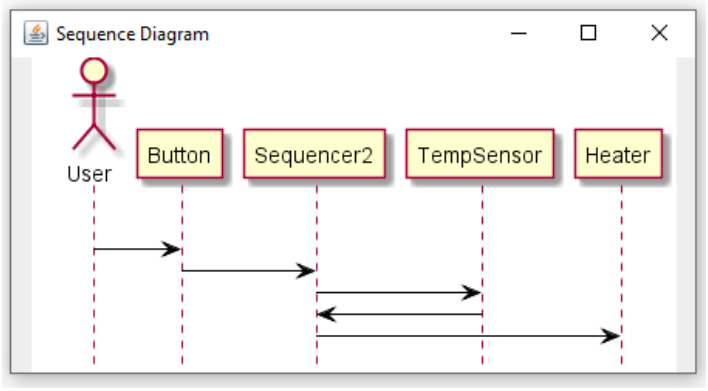


Figure 10.14 – Diagramme de séquence de l'application de réglage de température ambiante

Concernant la génération de la description à base de règles pour ce type d'architecture, nous avons proposé une solution (cf. section 6.4.5). Cependant, l'implémentation de cette solution n'est encore réalisée en totalité dans ICE. Actuellement, nous possédons une première version pour tester la faisabilité de la solution. Ainsi, nous avons pas encore fait d'expérience pour valider la génération d'une description à base de règles pour ce type d'architecture.

# 11 Analyse de la contribution

Ce chapitre présente une analyse de la contribution en s'appuyant sur les résultats des expérimentations présentés au chapitre précédent. Il commence par une évaluation de la couverture des exigences définies dans le chapitre 2. Ensuite, il discute l'apport de l'IDM pour répondre à nos objectifs. Enfin, il présente l'originalité de l'utilisation de l'IDM à destination d'un utilisateur final non expert.

## 11.1 Couverture des exigences

Les contributions proposées dans cette thèse visent à placer l'utilisateur au centre du système de composition, à lui permettre d'interagir avec son environnement ambiant et de garder le contrôle sur les services qui l'entourent. Ainsi, l'interface de contrôle ICE

— permet de visualiser une application émergente sous différentes formes pour aider l'utilisateur à comprendre le service offert par l'application et comment l'utiliser,

— permet de modifier l'application visualisée,

— laisse l'utilisateur décider de son déploiement.

Il s'agit à présent d'évaluer quelles exigences définies dans le chapitre 2 sont totalement ou partiellement satisfaites. Notre réponse à chaque exigence est évaluée de zéro à quatre + (c'est la même notation que celle utilisée pour l'état de l'art du chapitre 3). Elle est résumée dans le tableau 11.1.

Le premier groupe d'exigences (R1, R2) est relatif à la manière de présenter l'application pour permettre à l'utilisateur de la comprendre. Les transformations de modèles et les DSL nous ont permis de présenter une application selon plusieurs vues, exposant ainsi sa structure, sa fonctionnalité et ses usages. Les expérimentations ont montré que la solution actuelle fonctionne pour générer les modèles structurels de toutes les applications, indépen-

| Exigence | Avancement |
|---|---|
| [R1.1] Description fonctionnelle | ++ |
| [R1.2] Instructions d'utilisation | ++ |
| [R2.1] Intelligibilité de la description | +++ |
| [R2.2] Extensibilité de la description | + |
| [R3] Automatisation et composabilité | ++++ |
| [R4.1] Interventions de l'utilisateur | ++++ |
| [R4.2] Assistance à l'utilisateur | ++++ |
| [R5.1] Génération du retour d'informations | ++++ |
| [R5.2] Discrétion de la génération du retour d'informations | ++++ |

Table 11.1 – État d'avancement des réponses aux exigences

damment de leurs architectures. Les descriptions à base de règles, quant à elles, peuvent être générées pour les architectures de type Filtres et Tubes, parallèle ou séquentielle. Cependant, dans sa version actuelle, la solution ne permet pas la description à base de règles pour certains cas d'utilisation dans lesquels il existe des retours d'informations (cf. section 10.4.4). C'est parce que tous les types d'architectures ne peuvent pas être décrits à base de règles, que les exigences [R1.1] et [R1.2] ont été notées ++ (état moyen). [R2.1] a été notée +++ (état avancé), puisque la solution actuelle est capable d'adapter les modèles aux différents profils des utilisateurs, en particulier à un utilisateur moyen n'ayant pas de compétence dans le domaine de la programmation orientée composant. Jusqu'à présent nos expérimentations ont été menées sur des applications de taille moyenne (composées de moins d'une dizaine composants), mais aucune sur des applications composées d'une dizaine ou d'une quinzaine de composants. Aussi, [R2.2] a été évalué à +. L'extensibilité doit être éprouvée et améliorée dans de futurs travaux.

Le deuxième groupe d'exigences (R3) vise l'automatisation et la composabilité de la description. Actuellement, la solution est capable de construire et de présenter automatiquement à l'utilisateur la description de l'application. Par conséquent, elle répond totalement à l'exigence [R3] (d'où son état d'avancement à ++++).

Le troisième groupe d'exigences (R4, R5) concerne les interactions entre ICE et son environnement, à savoir OCE et l'utilisateur. Grâce aux outils et techniques de l'IDM utilisés pour développer ICE, ce dernier donne à l'utilisateur la possibilité d'éditer, d'accepter ou de rejeter une application après sa présentation (sachant qu'il a compris sa structure et la fonction du service offert) grâce aux DSL définis. En plus, les actions de modifications de l'application sont, premièrement, contrôlées à la volée par des règles d'édition existantes dans

les DSL et deuxièmement par des règles OCL (définis dans le métamodèle) lors de la phase de validation du modèle. Ceci permet d'aider l'utilisateur à construire des applications correctes en respectant la manière de connecter les services. En outre, les transformations et les comparaisons de modèles ont permis de capturer les interventions de l'utilisateur sans le surcharger et de les transformer en un modèle de feedback envoyé à OCE pour mettre à jour sa connaissance. Par conséquent, ICE a répondu totalement aux exigences [R4.1], [R4.2], [R5.1] et [R5.2] qui possèdent donc un état d'avancement de $++++$.

## 11.2 Apport et flexibilité de l'approche IDM

En se basant sur l'architecture de l'IDM proposée par l'OMG [Kent, 2002, OMG, 2015] (voir figure 1.7), nous avons défini plusieurs niveaux d'abstraction d'ICE : le métamodèle, les deux DSL et plusieurs méthodes de transformation de modèle. Ainsi, notre solution, basée sur les techniques de l'IDM, possède une grande flexibilité au niveau de sa construction.

En effet, le métamodèle permet de définir de manière explicite les concepts communs à toutes les applications qui peuvent être construites. En outre, grâce à l'IDM, les descriptions (structurelles ou sémantiques) de l'application sont facilement obtenues et ajustées à un environnement humain particulier, à travers la simplicité de changer et d'adapter les DSL et les méthodes de transformation de modèle par rapport au profil de l'utilisateur. Par exemple, dans le cas où un utilisateur préfère une autre forme de présentation structurelle de l'application, il faut tout simplement prendre un des DSL existants et modifier seulement la manière de représenter chaque élément : composants, services (requis et fournis) et connexions. La modification du métamodèle pour intégrer une ontologie pour décrire les services et la traduction en langage naturel offrent une autre possibilité d'adaptation (voir, plus loin, la discussion sur les suites possibles de ce travail).

De manière générale, l'IDM permet de gérer l'intégralité du cycle de vie d'un système à développer comme un processus de production, de raffinement et d'intégration de modèles [Azaiez, 2007], en commençant tout d'abord par la phase de modélisation, puis la phase de présentation et de modification et enfin la phase de déploiement de l'application. Cette approche a apporté une solution aux problèmes posés dans cette thèse :

— Au moyen d'ICE, l'utilisateur peut prendre connaissance des applications qui émergent, grâce à la présentation de leurs modèles sous la forme de plusieurs vues et de descriptions différentes.

— L'utilisateur est placé au centre du système dans son ensemble (qui inclut l'environnement ambiant, OCE et ICE), en lui donnant le rôle principal et le contrôle nécessaire pour lui permettre de prendre la décision de modifier, d'accepter ou de rejeter l'application proposée. Ainsi les responsabilités sont réparties entre le moteur OCE qui propose des applications et l'utilisateur qui contrôle.

— Le feedback implicite de l'utilisateur est capturé et transmis à OCE. Sans contribution supplémentaire de l'utilisateur, OCE apprend à partir de ce feedback et prend des décisions pertinentes [Younes et al., 2019a].

Les modèles produits à chaque niveau d'abstraction deviennent alors des artefacts pouvant être pleinement exploités au niveau du cycle de développement, contrairement aux approches de développement classiques où les modèles permettaient uniquement de documenter les applications en cours de développement.

## 11.3 L'IDM pour présenter les applications à l'utilisateur final

Dans la plupart des travaux existants, les éditeurs qui offrent aux utilisateurs finaux la possibilité de construire ou de modifier leurs applications sont développés de manière ad-hoc. Les modifications sont limitées au choix de modèles alternatifs proposés par l'éditeur. Dans ce cas, les utilisateurs sont assistés mais ne possèdent qu'un contrôle limité sur les applications développées.

L'approche à base d'IDM proposée dans cette thèse permet d'éviter cette situation en exploitant les modèles d'applications de manière productive. Ceci signifie que les applications sont assemblées, modifiées ou transformées en utilisant un métamodèle, des DSL et en exécutant des règles de transformation explicitement exprimées. La solution proposée permet à l'utilisateur final, qui peut être un utilisateur moyen non expert dans le domaine de programmation, d'utiliser les outils de l'IDM pour configurer ses applications, à travers une représentation personnalisée correspondante. En outre, il n'existe pas des travaux dans l'état de l'art qui visent à présenter des applications à des utilisateurs non experts, à leur donner le contrôle nécessaire pour configurer directement leurs applications et à les assiter dans cette tâche. Par ailleurs, comme mentionné dans la section précédente, l'IDM facilite le passage d'une forme de représentation à une autre, ce qui permet d'adapter la présentation à l'utilisateur. Notre utilisation de l'IDM est différente des solutions existantes où l'IDM est utilisée par des experts et des développeurs.

# Conclusion et perspectives

## Conclusion

Les systèmes ambiants, avec leur ouverture et leur dynamique, permettent difficilement de concevoir et de mettre en œuvre des logiciels en suivant les cycles de développement traditionnels dans lesquels les applications sont construites à partir d'un besoin explicite de l'utilisateur et d'un ensemble de composants logiciels connus. Les problèmes viennent en particulier de l'instabilité des dispositifs et des composants qui peuvent apparaître et disparaître sans que cette dynamique ne soit nécessairement anticipée. De plus, le besoin de l'utilisateur n'est pas stable non plus. Il peut varier rapidement en fonction de la situation d'une manière imprévisible. Pour répondre à ces problèmes, notre équipe explore une solution alternative dans laquelle un moteur de composition "intelligent" et autonome construit et fait émerger, en fonction des opportunités, des applications que l'utilisateur n'a pas explicitement demandées ni attendues.

Au centre des systèmes ambiants, l'utilisateur doit pouvoir bénéficier des bons services au bon moment, avec le moins d'effort possible de sa part. Pour que le moteur intelligent crée des applications pertinentes c'est-à-dire adaptées à sa situation, à ses préférences et à ses compétences, son implication est néanmoins indispensable. Ainsi, il doit pouvoir intervenir pour contrôler l'émergence, c'est-à-dire accepter ou rejeter les applications construites par le moteur. Il doit aussi pouvoir les modifier et contribuer ainsi à la configuration des applications, donc de son environnement. Dans cette tâche de développement, cet utilisateur doit être assisté, en particulier pour garantir la correction des applications construites.

Dans ce contexte, plusieurs problèmes nous étaient posés. Le premier était d'informer l'utilisateur des applications émergentes proposées par le système intelligent et de les présenter d'une manière intelligible. Le deuxième était de donner un rôle et des moyens à l'uti-

lisateur, au centre du système, pour intervenir et en être l'acteur principal.

Pour répondre à ces problèmes, une approche originale est proposée dans cette thèse : nous montrons comment les techniques de l'IDM avec la définition d'un métamodèle et de DSL permettent à la fois de fournir à l'utilisateur des vues personnalisées des applications, et de produire les outils qui permettent de modifier les applications ou même de les construire à partir de rien. Concrètement, le résultat est un éditeur graphique appelé ICE (Interactive Control Environment) qui est adapté au profil des utilisateurs. Ainsi, l'IDM supporte à la fois la production de matériel descriptif et la construction contrôlée des applications. En outre, en comparant et en transformant les modèles, un retour d'information est capturé et fourni au moteur intelligent pour alimenter son processus d'apprentissage.

En utilisant les techniques et les outils offerts par l'IDM, nous avons mis en œuvre et expérimenté une version pleinement opérationnelle d'ICE. Ce dernier, grâce aux DSL, permet à son utilisateur de visualiser l'application sous forme de deux représentations structurelles différentes : un diagramme de composants pour les utilisateurs experts dans le domaine de la programmation et une représentation à base de pictogrammes pour les utilisateurs non experts. En outre, ICE transforme le modèle structurel de l'application pour offrir à l'utilisateur, d'une part, un diagramme de séquence de l'application qui explicite les interactions entre les composants, et d'autre part, une description à base de règles qui décrit le service offert par l'application et comment l'utiliser. Ainsi, à travers les descriptions structurelles et sémantiques de l'application, l'utilisateur peut comprendre l'application qui est proposée, et au besoin la modifier. Cette étape est contrôlée par des règles OCL et les DSL qui guident l'utilisateur dans la construction des applications. Par ailleurs, le résultat des interventions de l'utilisateur est capturé par ICE sans surcharger l'utilisateur. Ces actions de l'utilisateur sont envoyées en tant que feedback à OCE qui les utilise pour apprendre le besoin de l'utilisateur en situation. Par ailleurs, l'utilisation d'une approche basée sur l'IDM apporte de la flexibilité à notre solution en permettant de changer le langage de description et d'adapter les présentations au profil de l'utilisateur.

A ce stade, ICE possède certaines limites dont le traitement fait l'objet de travaux actuellement initiés ou en projet. Les descriptions structurelles et sémantiques doivent être davantage testées et améliorées pour supporter les applications possédant une architecture plus complexe ou qui consistent en un nombre de composants élevé. Ceci permettrait de mieux évaluer la flexibilité de l'approche. De plus, il faudrait mieux évaluer l'intelligibilité

et l'extensibilité des descriptions, en particulier auprès des utilisateurs non experts. En fonction de leurs profils, d'autres vues pourraient être fournies. Une piste possible serait de se rapprocher des langages de blocs [Bau et al., 2017].

Dans ce qui suit, nous discutons de quelques perspectives.

## Perspectives

### Amélioration de la description de l'application

Dans la solution actuelle, la génération de la description à base de règles fonctionne pour des cas d'utilisation simples où le nombre de composants est limité. De plus, concernant les applications où existe un retour de données, la génération de la description à base de règles ne fonctionne pas (cf. section 10.4.4). Nous avons donc défini un travail qui vise à améliorer le processus de génération de description à base de règles, et surtout à le tester et à le valider sur des applications qui consistent en un nombre plus important de composants (une dizaine par exemple).

Concernant le diagramme de séquence, actuellement, grâce à notre transformation de modèle, ICE est capable de générer le diagramme de toutes les applications présentées, indépendemment du nombre de composants qui les composent ou de leurs architectures. Cependant, nous visons aussi à améliorer le diagramme de séquence de l'application, en ajoutant plus d'informations sur le diagramme affiché à l'utilisateur. Par exemple, faire afficher les conditions gérées par les règles avec les variables échangées. Ceci offrira à l'utilisateur plus d'informations sur le mode de fonctionnement de son application.

En outre, plusieurs vues descriptives de l'application pourront être ajoutées à ICE. Par exemple, une représentation de l'application sous forme d'un texte ou d'autre choix possibles, selon la demande de l'utilisateur. Cette tâche est réalisable et ne requiert pas de travail de développement trop important, de codage et de ressources, grâce aux avantages de l'IDM (la définition de DSL en particulier).

### Description textuelle

Nous avons testé la faisabilité d'une description textuelle de l'application. Une fois les règles générées, ICE est capable, en utilisant les attributs des descriptions des services, de transformer les règles en un texte libre qui décrit l'application. La figure 11.1 montre

un exemple de génération de description d'une application composée d'un interrupteur et d'une lampe. Dans cette application, la combinaison de règles donne le résultat suivant :

**Résultat .** onButtonPressed & STATE==ON $\Longrightarrow$ STATE==OFF

onButtonPressed & STATE==OFF $\Longrightarrow$ STATE==ON

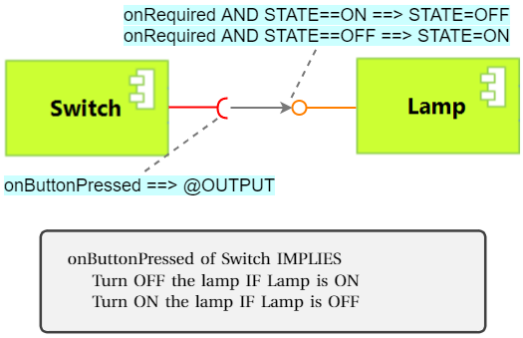


Figure 11.1 – Description textuelle de l'application émergente

La dernière fenêtre affichée présente la description textuelle de l'application. Pour l'obtenir, le déclencheur principal, ou partie "Evénement" de la règle, est associé avec le nom du composant : "*onButtonPressed of Switch*". Ensuite, pour chaque condition existante, le processus la transforme en version textuelle en commençant par la partie "Action" de la règle puis la "Condition". Par exemple, "*STATE==ON THEN STATE=OFF*" est transformée en "*Turn OFF Lamp IF Lamp is ON*". Pour obtenir le terme "*Turn OFF*" l'algorithme remplace le contenu de l'attribut *Function* (valeur : Turn ON/OFF) existant dans la description du service par la bonne valeur de l'état de la Lampe.

Les descriptions étant définies par nous-mêmes, elles sont homogènes. Cependant, cette méthode présente des problèmes lorsque l'application consiste en un grand nombre de composants ou si les descriptions proviennent de plusieurs fournisseurs (ce qui devrait généralement être le cas).

Pour rendre la description de l'application plus intelligible surtout par les utilisateurs non experts, une première idée est que les différents fournisseurs de services utilisent une même ontologie que notre équipe a définie pour décrire les services implémentés par de composants logiciels [Alary et al., 2020a]. Ceci rendrait les descriptions homogènes et faciliterait la génération du texte. Alors, on pourrait utiliser les technologies et les méthodes de traitement du langage naturel (NLP) pour générer un texte structuré et pertinent, en se basant sur l'ontologie décrivant les composants et leurs services et sur la description à base de règles.

### Utilisation des ontologies pour améliorer la description à base de règles

A ce stade, pour générer une description à base de règles d'une application émergente, notre solution se base sur un langage de description que nous avons proposé dans le but de tester la faisabilité de l'approche et de simplifier les combinaisons de règles. Nous avons réussi à obtenir un premier résultat afin de valider la contribution. Cependant, cette solution peut être améliorée.

Ainsi, on peut remplacer dans le métamodèle les éléments de la section de description et des règles par des éléments se basant sur des ontologies ([McGuinness et al., 2004]). Notre équipe a déjà commencé à travailler sur cette approche. Dans ce travail, les auteurs ont proposés un ontologie nommée "Comp-O" qui est une extension d'OWL-S [W3C, 2004] pour les services implémentés par des assemblages de composants logiciels [Alary et al., 2020a, Alary et al., 2020b]. Comp-O prend en charge la description des services requis et la combinaison des descriptions afin de générer automatiquement la description des services "composites" à partir des descriptions unitaires des composants utilisés dans l'assemblage. En outre, Comp-O assiste le développeur dans la sélection des composants à assembler (mise en correspondance entre les interfaces des services requis et fournis) pour construire un assemblage de composants. L'intégration de Comp-O dans notre système pourrait aussi permettre au moteur OCE de proposer des applications plus pertinentes accompagnées de leurs descriptions.

### Pliage et dépliage de la présentation structurelle et sémantique

L'objectif est de permettre à l'utilisateur de connaître plus ou moins de détails sur la construction de son application et comment elle fonctionne. Ceci serait possible en lui donnant la possibilité de plier et de déplier la description structurelle de l'application et en lui permettant d'accéder à une représentation à plusieurs niveaux d'abstraction.

Voici les détails du pliage et du dépliage de l'application :

1. Tout d'abord l'application serait présentée en montrant certains composants sélectionnés, par exemple les composants d'interaction avec l'utilisateur ou avec l'environnement physique.
2. L'utilisateur, pour obtenir plus d'information, pourrait déplier la représentation de l'application (ou "zoomer") et obtenir une représentation qui visualise aussi les composants

et services intermédiaires initialement cachés.

3. Inversement, pour réduire la description structurelle, l'utilisateur pourrait plier son application pour visualiser moins de composants.
4. Les descriptions structurelles et sémantiques évolueraient avec chaque pliage ou dépliage.

L'objectif de mieux guider l'utilisateur dans sa compréhension des services offerts par l'application et comment elle fonctionne.

### Animation du modèle

Une autre piste, que nous envisageons d'explorer, est l'animation de modèles [Combemale et al., 2008]. GEMOC permet d'animer un modèle [Gemoc Initiative, 2020] en montrant les différentes étapes de son exécution. D'une autre manière, il permet de visualiser les différents appels de services et les messages échangés. De plus, si le modèle est modifié, l'animation s'adapte aux nouveaux changements. En outre, cet outil permet de sauvegarder tous les modifications faites durant la phase d'animation, ce qui permet à l'utilisateur de directement rétablir un ancien modèle si le nouveau ne répond pas à son besoin (d'après l'animation de son exécution).

Ainsi, l'idée est de configurer ICE pour être capable d'animer les modèles d'application présentés pour donner à l'utilisateur une vue complémentaire de l'application et lui faire mieux comprendre le service offert et le fonctionnement.

### Déploiement d'ICE (Standalone Editor)

Actuellement, ICE consiste en un projet EMF qui définit le métamodèle et un projet Sirius qui définit les DSL, l'éditeur et les transformations de modèle gérés par ICE. Pour accéder à l'éditeur, l'utilisateur doit lancer tout d'abord le projet EMF du métamodèle, puis lancer le projet Sirius en mode *Runtime Eclipse Application*.

Par conséquent, pour déployer ICE sur un poste utilisateur, il faut suivre plusieurs étapes :

1. La première consiste à installer GEMOC et le configurer avec la version de Java compatible avec le projet ICE.

2. La deuxième consiste à importer les projets d'ICE et les configurer pour que l'utilisateur puisse l'utiliser.
3. La troisième consiste à configurer M[OI]CE pour connecter ICE à OCE.

Finalement, le déploiement d'ICE est une tâche délicate qui implique l'utilisation et donc le déploiement d'un certain nombre d'outils. Une perspective serait de fournir ICE comme une application autonome, prête à être déployée sans installation ou configuration intermédiaires, ce que l'environnement GEMOC ne permet pas de faire actuellement.

### Apprentissage de modèles

Fondamentalement, l'apprentissage automatique [Mitchell, 1997] vise à construire des connaissances ou des modèles. Ces derniers sont construits automatiquement et de manière itérative dans la phase d'apprentissage à partir de données d'apprentissage. L'objectif est d'améliorer le comportement et les performances du système qui apprend quand il sera dans une phase ultérieure d'exploitation.

Le chapitre 7 a présenté comment la comparaison des modèles supporte la génération d'un retour d'information pour l'apprentissage d'OCE. Ici, nous allons au-delà des exigences définies dans le chapitre 2 : nous pouvons profiter des facilités de l'IDM pour concevoir un mode d'apprentissage complémentaire.

Pour contribuer à améliorer la pertinence des décisions d'OCE, l'idée est de lui fournir des plans d'assemblages prêts à l'emploi. Le problème est d'injecter ces plans dans OCE, qui n'a pas été conçu pour manipuler de tels artefacts et qui ne crée pas des applications en s'appuyant sur des plans (au contraire, OCE fabrique des plans d'assemblage qui sont ensuite présentés dans ICE et manipulés par l'utilisateur). Le principe de notre proposition est de transformer les plans des applications (plans d'assemblage), lorsqu'ils sont acceptés par l'utilisateur, en des composants particuliers appelés connecteurs. Ces connecteurs implémentent ces plans, en faisant abstraction des composants qui participent à l'assemblage. Un connecteur met en œuvre le modèle de conception "Mediateur" [Gamma et al., 1994] qui centralise la logique d'interaction entre les composants : il achemine les requêtes d'un service appelant vers un service appelé et achemine les résultats dans la direction opposée. Pour cela, le connecteur est implanté par un composant qui rassemble tous les services fournis et requis de tous les composants impliqués dans le plan d'assemblage. Dans le plan d'assemblage équivalent basé sur un connecteur, les liens directs entre les composants sont

remplacés par des liens entre chaque composant et le connecteur. La figure 11.2 montre l'assemblage à base de connecteurs de l'application d'éclairage ambiant : il est équivalent à celui de la figure 10.10 mais l'interaction entre les composants est centralisée dans le connecteur.

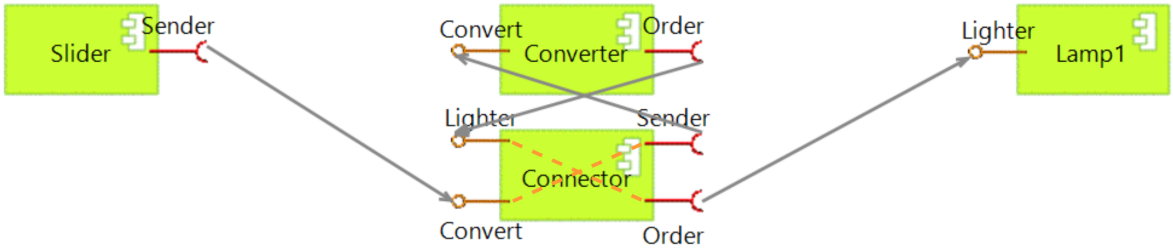


Figure 11.2 – Assemblage à base de connecteur

Nous avons expérimenté cette proposition. Pour générer le connecteur à partir d'un plan d'assemblage, *EMF model factory* est utilisée. Un modèle du composant du connecteur est créé avec tous les services fournis et requis impliqués dans l'ensemble de l'assemblage. Si nécessaire, une transformation de modèle à modèle peut reconstruire le modèle de l'application basé sur les connecteurs tel qu'il est affiché dans la figure 11.2.

Dans un deuxième temps, une transformation de modèle en texte permet de produire le code d'implémentation du connecteur (code Java dans notre cas) et un script pour son déploiement dans l'environnement. Après déploiement, il peut être détecté et traité par OCE comme n'importe quel composant logiciel ordinaire.

# Bibliographie

[Abrahão et al., 2017] Abrahão, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., and Whittle, J. (2017). User experience for model-driven engineering : Challenges and future directions. In *20th ACM/IEEE Int. Conf. on MDE Languages and Systems*, pages 229–236. IEEE.

[Alary et al., 2020a] Alary, G., Hernandez, N., Arcangeli, J.-P., Trouilhet, S., and Bruel, J.-M. (2020a). Comp-O : an OWL-S Extension for Composite Service Description. In *International Conference on Knowledge Engineering and Knowledge Management (EKAW)*. Springer. `https://ekaw2020.inf.unibz.it/wp-content/uploads/2020/09/59-Hernandez.pdf`.

[Alary et al., 2020b] Alary, G., Hernandez, N., Arcangeli, J.-P., Trouilhet, S., and Bruel, J.-M. (2020b). Using Comp-O to Build and Describe Component-Based Services. In *19th Int. Semantic Web Conference (ISWC)*. Demo track, To appear.

[Ankolekar et al., 2002] Ankolekar, A., Burstein, M., Hobbs, J. R., Lassila, O., Martin, D., McDermott, D., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T., et al. (2002). Daml-s : Web service description for the semantic web. In *International semantic web conference*, pages 348–363. Springer.

[Antoniou and Van Harmelen, 2004] Antoniou, G. and Van Harmelen, F. (2004). Web ontology language : Owl. In *Handbook on ontologies*, pages 67–92. Springer.

[Azaiez, 2007] Azaiez, S. (2007). *Approche dirigée par les modèles pour le développement de systèmes multi-agents*. Theses, Université de Savoie.

[Bach and Scapin, 2003] Bach, C. and Scapin, D. (2003). Adaptation of ergonomic criteria to human-virtual environments interactions. In *Proc. of Interact'03*, pages 880–883. IOS Press.

[Baciková et al., 2013] Baciková, M., Porubän, J., and Lakatos, D. (2013). Defining Domain Language of Graphical User Interfaces. In *2nd Symposium on Languages, Applications and*

*Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 187–202.

[Barricelli et al., 2019] Barricelli, B. R., Cassano, F., Fogli, D., and Piccinno, A. (2019). End-user development, end-user programming and end-user software engineering : A systematic mapping study. *Journal of Systems and Software*, 149 :101 – 137.

[Bartalos and Bieliková, 2012] Bartalos, P. and Bieliková, M. (2012). Automatic dynamic web service composition : A survey and problem formalization. *Computing and Informatics*, 30(4) :793–827.

[Bau et al., 2017] Bau, D., Gray, J., Kelleher, C., Sheldon, J., and Turbak, F. (2017). Learnable programming : Blocks and beyond. *Communication of the ACM*, 60(6) :72–80.

[Berndtsson and Mellin, 2009] Berndtsson, M. and Mellin, J. (2009). *ECA Rules*, pages 959–960. Springer US, Boston, MA.

[Bézivin, 2006] Bézivin, J. (2006). *Model Driven Engineering : An Emerging Technical Space*, pages 36–64. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Bubble, 2012] Bubble (2012). Bubble : The most powerful no-code platform. `https://bubble.io/`. Online : accessed 20/10/2020.

[Bucchiarone et al., 2019] Bucchiarone, A., Cicchetti, A., and Marconi, A. (2019). Exploiting Multi-level Modelling for Designing and Deploying Gameful Systems. In *22nd ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS 2019)*, pages 34–44. IEEE.

[Carmien and Fischer, 2008] Carmien, S. P. and Fischer, G. (2008). Design, adoption, and assessment of a socio-technical environment supporting independence for persons with cognitive disabilities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 597–606.

[Chen et al., 2015] Chen, X., Li, A., Guo, W., Huang, G., et al. (2015). Runtime model based approach to IoT application development. *Frontiers of Computer Science*, 9(4) :540–553.

[Chindenga et al., 2017] Chindenga, E., Scott, M. S., and Gurajena, C. (2017). Semantics Based Service Orchestration in IoT. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT'17, pages 7 :1–7 :7, New York, NY, USA. ACM.

[Ciccozzi and Spalazzese, 2016] Ciccozzi, F. and Spalazzese, R. (2016). MDE4IoT : Supporting the Internet of Things with Model-Driven Engineering. In *International Symposium on Intelligent and Distributed Computing*, pages 67–76. Springer.

[CNRTL, 2012] CNRTL (2012). Opportunisme. https://www.cnrtl.fr/definition/opportunisme. Online : accessed 20/10/2020.

[Combemale et al., 2008] Combemale, B., Crégut, X., Giacometti, J.-P., Michel, P., and Pantel, M. (2008). Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In *4th European Congress on Embedded Real-Time Software (ERTS 2008)*. SIA.

[Combemale et al., 2016] Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J., and Vojtisek, D. (2016). *Engineering modeling languages : Turning domain knowledge into tools*. Chapman & Hall.

[Coutaz, 2007] Coutaz, J. (2007). Meta-user interfaces for ambient spaces. In Coninx, K., Luyten, K., and Schneider, K. A., editors, *Task Models and Diagrams for Users Interface Design*, pages 1–15, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Coutaz and Crowley, 2016] Coutaz, J. and Crowley, J. (2016). A First-Person Experience with End-User Development for Smart Homes. *IEEE Pervasive Computing*, 15 :26 – 39.

[Cox, 1986] Cox, B. J. (1986). Object-oriented programming : an evolutionary approach. *274 p. : ill Reading, Mass. : Addison-Wesley Pub. Co., 1986. includes index*.

[Degas et al., 2016] Degas, A., Trouilhet, S., Arcangeli, J.-P., Calvary, G., Coutaz, J., Lavirotte, S., and Tigli, J.-Y. (2016). Opportunistic Composition of Human-Computer Interactions in Ambient Spaces. In *Workshop on Smart and Sustainable City (Smart World Congress 2016 & Int. Conf. IEEE UIC 2016)*, pages 998–1005. IEEE Computer Society.

[Eclipse, 2020a] Eclipse (2020a). Acceleo. https://www.eclipse.org/acceleo/. Online : accessed 20/10/2020.

[Eclipse, 2020b] Eclipse (2020b). Acceleo query language. https://www.eclipse.org/acceleo/documentation/. Online : accessed 20/10/2020.

[Eclipse, 2020c] Eclipse (2020c). Eclipse ide for a java integrated development environment. https://www.eclipse.org/ide/. Online : accessed 20/10/2020.

[Eclipse, 2020d] Eclipse (2020d). Eclipse modeling framework (emf). https://www.eclipse.org/modeling/emf/. Online : accessed 20/10/2020.

[Eclipse, 2020e] Eclipse (2020e). Eclipse modeling project. https://www.eclipse.org/modeling/. Online : accessed 20/10/2020.

[Eclipse, 2020f] Eclipse (2020f). Ecoretools - graphical modeling for ecore. https://www.eclipse.org/ecoretools/. Online : accessed 20/10/2020.

[Eclipse, 2020g] Eclipse (2020g). Sirius. https://www.eclipse.org/sirius/overview.html. Online : accessed 20/10/2020.

[Evers et al., 2014] Evers, C., Kniewel, R., Geihs, K., and Schmidt, L. (2014). The user in the loop : Enabling user participation for self-adaptive applications. *Future Generation Computer Systems*, 34 :110–123.

[Fanjiang et al., 2017] Fanjiang, Y., Syu, Y., Ma, S., and Kuo, J. (2017). An overview and classification of service description approaches in automated service composition research. *IEEE Transaction on Services Computing*, 10(2) :176–189.

[Faure et al., ] Faure, M., Fabresse, L., Huchard, M., Urtado, C., and Vauttier, S. User-defined scenarios in ubiquitous environments : Creation, execution control and sharing. Citeseer.

[Floch et al., 2013] Floch, J., Frà, C., Fricke, R., Geihs, K., Wagner, M., Lorenzo, J., Soladana, E., Mehlhase, S., Paspallis, N., Rahnama, H., et al. (2013). Playing music—building context-aware and self-adaptive mobile applications. *Software : Practice and Experience*, 43(3) :359–388.

[Fowler, 2010] Fowler, M. (2010). *Domain-specific languages*. Pearson Education.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley. Pearson Education.

[Geihs et al., 2009] Geihs, K., Barone, P., Eliassen, F., Floch, J., Fricke, R., Gjorven, E., Hallsteinsen, S., Horn, G., Khan, M. U., Mamelli, A., et al. (2009). A comprehensive solution for application-level adaptation. *Software : Practice and Experience*, 39(4) :385–422.

[Gemoc Initiative, 2012] Gemoc Initiative (2012). Gemoc studio. http://gemoc.org/. Online : accessed 20/10/2020.

[Gemoc Initiative, 2020] Gemoc Initiative (2020). Animation de modèle avec gemoc studio. https://download.eclipse.org/gemoc/docs/nightly/userguide-mw-execute-animate-debug-models.html. Online : accessed 20/10/2020.

[Ghiani et al., 2017] Ghiani, G., Manca, M., Paternò, F., and Santoro, C. (2017). Personalization of context-dependent applications through trigger-action rules. *ACM Trans. on Computer-Human Interaction*, 24(2).

[Ghiani et al., 2009] Ghiani, G., Paternò, F., and Spano, L. D. (2009). Cicero designer : an environment for end-user development of multi-device museum guides. In *International Symposium on End User Development*, pages 265–274. Springer.

[Gil et al., 2016] Gil, M., Pelechano, V., Fons, J., and Albert, M. (2016). Designing the Human in the Loop of Self-Adaptive Systems. In *$10^{th}$ Int. Conf. on Ubiquitous Computing and Ambient Intelligence*, pages 437–449. Springer.

[Gomez et al., 2006] Gomez, J. M., Han, S., Toma, I., Sapkota, B., and Garcia-Crespo, A. (2006). A Semantically-enhanced Component-based Architecture for Software Composition. In *Int. Multi-Conf. on Computing in the Global Information Technology (ICCGI'06)*, pages 43–47.

[Google, 2014] Google (2014). Appsheet : The intelligent no-code platform. `https://www.appsheet.com/`. Online : accessed 20/10/2020.

[Hahn and Fischer, 2007] Hahn, C. and Fischer, K. (2007). Service composition in holonic multiagent systems : Model-driven choreography and orchestration. In Mařík, V., Vyatkin, V., and Colombo, A. W., editors, *Holonic and Multi-Agent Systems for Manufacturing*, pages 47–58, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Hamoui, 2010] Hamoui, M. F. (2010). *U niversité M ontpellier II*. PhD thesis, École des Mines d'Alès.

[Hurault et al., 2009] Hurault, A., Camillo, F., Daydé, M., Guivarch, R., Pantel, M., Puglisi, C., and Astsatryan, H. (2009). Semantic description of services : issues and examples. Computer Science and Information Technologies, Yerevan (Arménia).

[Isbell et al., 2006] Isbell, C. L., Kearns, M., Singh, S., Shelton, C. R., Stone, P., and Kormann, D. (2006). Cobot in lambdamoo : An adaptive social statistics agent. *Autonomous Agents and Multi-Agent Systems*, 13(3) :327–354.

[Janisch, 2010] Janisch, S. (2010). *Behaviour and Refinement of Port-Based Components with Synchronous and Asynchronous Communication*. PhD thesis, LMU Munich.

[Karami et al., 2016] Karami, A. B., Fleury, A., Boonaert, J., and Lecoeuche, S. (2016). User in the Loop : Adaptive Smart Homes Exploiting User Feedback—State of the Art and Future Directions. *Information*, 7(2).

[Kent, 2002] Kent, S. (2002). Model driven engineering. In *Integrated Formal Methods*, pages 286–298. Springer.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1) :41–50.

[Klusch, 2008] Klusch, M. (2008). Semantic Web Service Description. In *CASCOM : Intelligent Service Coordination in the Semantic Web*, pages 31–57. Birkhäuser Basel.

[Koussaifi, 2019] Koussaifi, M. (2019). User-oriented description of emerging services in ambient systems. In *Service-Oriented Computing - ICSOC 2019 Workshops - WESOACS, ASOCA, ISYCC, TBCE, and STRAPS, Toulouse, France, October 28-31, 2019, Revised Selected Papers*, volume 12019 of *Lecture Notes in Computer Science*, pages 259–265. Springer. https://doi.org/10.1007/978-3-030-45989-5_21.

[Koussaifi et al., 2020] Koussaifi, M., Arcangeli, J.-P., Trouilhet, S., and J.-M., B. (2020). Putting the End-User in the Loop in Smart Ambient Systems : an Approach based on Model-Driven Engineering. Technical report IRIT/RR–2020–06–FR, IRIT. https://www.irit.fr/~Sylvie.Trouilhet/publiInfo/IRIT_RR_2020_06_FR.pdf.

[Koussaifi et al., 2019] Koussaifi, M., Trouilhet, S., Arcangeli, J.-P., and Bruel, J.-M. (2019). Automated user-oriented description of emerging composite ambient applications. In *Proc. of the 31st Int. Conf. on Software Engineering and Knowledge Engineering*, pages 473–478. https://hal.archives-ouvertes.fr/hal-02467549/document.

[Koussaifi et al., 2018] Koussaifi, M., Younes, W., Adreit, F., Arcangeli, J.-P., Bruel, J.-M., and Trouilhet, S. (2018). Emergence of Composite Services in Smart Environments. 8th EuroScience Open Forum (ESOF 2018). Poster, https://hal.archives-ouvertes.fr/hal-02280356/file/koussaifi_22488.pdf.

[Koussaïfi et al., 2018] Koussaïfi, M., Trouilhet, S., Arcangeli, J.-P., and Bruel, J.-M. (2018). Ambient intelligence users in the loop : Towards a model-driven approach. In *Software Technologies : Applications and Foundations*, pages 558–572. Springer. https://hal.archives-ouvertes.fr/hal-01815481/document.

[Lawler and Howell-Barber, 2007] Lawler, J. P. and Howell-Barber, H. (2007). *Service-oriented architecture : SOA strategy, methodology, and technology*. CRC Press.

[Letondal and Mackay, 2004] Letondal, C. and Mackay, W. E. (2004). Participatory programming and the scope of mutual responsibility : balancing scientific, design and software commitment. In *Proceedings of the eighth conference on Participatory design : Artful integration : interweaving media, materials and practices-Volume 1*, pages 31–41.

[Lewis and Fowler, 2014] Lewis, J. and Fowler, M. (2014). Microservices.

[Liangzhao Zeng et al., 2004] Liangzhao Zeng, Benatallah, B., Ngu, A. H. H., Dumas, M., Kalagnanam, J., and Chang, H. (2004). Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5) :311–327.

[Ludewig, 2003] Ludewig, J. (2003). Models in software engineering – an introduction. *Software and Systems Modeling*, 2(1) :5–14.

[McGuinness et al., 2004] McGuinness, D. L., Van Harmelen, F., et al. (2004). Owl web ontology language overview. *W3C recommendation*, 10(10) :2004.

[McIlroy et al., 1968] McIlroy, M. D., Buxton, J., Naur, P., and Randell, B. (1968). Mass-produced software components. In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pages 88–98.

[Medjahed et al., 2003] Medjahed, B., Bouguettaya, A., and Elmagarmid, A. K. (2003). Composing web services on the semantic web. *The VLDB Journal*, 12(4) :333–351.

[MIT, 2020] MIT (2020). App inventor mit. `http://appinventor.mit.edu/`. Online : accessed 20/10/2020.

[Mitchell, 1997] Mitchell, T. (1997). *Machine Learning*. McGraw-Hill.

[Nestler et al., 2010] Nestler, T., Feldmann, M., Hübsch, G., Preußner, A., and Jugel, U. (2010). The servface builder - a wysiwyg approach for building service-based applications. In Benatallah, B., Casati, F., Kappel, G., and Rossi, G., editors, *Web Engineering*, pages 498–501, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Newcomer and Lomow, 2005] Newcomer, E. and Lomow, G. (2005). *Understanding SOA with Web services*. Addison-Wesley.

[Newman et al., 2003] Newman, M., Lin, J., Hong, J., and Landay, J. (2003). Denim : An informal web site design tool inspired by observations of practice. *Human-Computer Interaction*, 18 :259–324.

[OMG, 2015] OMG (2015). Qvt transformation specification. `https://www.omg.org/spec/QVT/1.2/`. Online : accessed 20/10/2020.

[OMG, 2020a] OMG (2020a). Common object request broker architecture. `https://corba.org/`. Online : accessed 20/10/2020.

[OMG, 2020b] OMG (2020b). Unified mpodeling language (uml). `https://www.uml-diagrams.org/`. Online : accessed 20/10/2020.

[Oracle, 2020] Oracle (2020). Java remote method invocation (rmi). `https://docs.oracle.com/javase/tutorial/rmi/index.html`. Online : accessed 20/10/2020.

[Paternò, 2013] Paternò, F. (2013). End User Development : Survey of an Emerging Field for Empowering People. *ISRN Software Engineering*, 2013.

[PlantUML, 2020] PlantUML (2020). Plantuml in a nutshell. https://plantuml.com/. Online : accessed 20/10/2020.

[Rainbow, 2006] Rainbow, I3S, U. o. N. S. A. (2006). Wcomp. http://www.wcomp.fr/wikiwcomp. Online : accessed 20/10/2020.

[Rousse, 2007] Rousse, N. (2007). Model driven architecture (mda). http://www.modelia.org/html/9_fichesTechniques/8000_IDMcahier.pdf. Online : accessed 20/10/2020.

[Rushby, 1995] Rushby, J. (1995). Model checking and other ways of automating formal methods. *Position paper for panel on Model Checking for Concurrent Programs, Software Quality Week, San Francisco*.

[Schmidt, 2006] Schmidt, D. C. (2006). Guest editor's introduction : Model-driven engineering. *Computer*, 39(2) :25–31.

[Schmidt, 2006] Schmidt, D. C. (2006). Model-Driven Engineering. *Computer*, 39(2) :25–31.

[Sendall and Kozaczynski, 2003] Sendall, S. and Kozaczynski, W. (2003). Model transformation : the heart and soul of model-driven software development. *IEEE Software*, 20(5) :42–45.

[Sheng et al., 2014] Sheng, Q. Z., Qiao, X., Vasilakos, A. V., Szabo, C., Bourne, S., and Xu, X. (2014). Web services composition : A decade's overview. *Information Sciences*, 280 :218 – 238.

[Sommerville, 2016] Sommerville, I. (2016). Component-based software engineering. In *Software Engineering*, chapter 16, pages 464–489. Pearson Education, $10^{th}$ edition.

[Sutton and Barto, 2018] Sutton, R. and Barto, A. (2018). *Reinforcement Learning : An Introduction*. MIT Press.

[Sutton and Barto, 2011] Sutton, R. S. and Barto, A. G. (2011). Reinforcement learning : An introduction.

[Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component software : beyond object-oriented programming*. Pearson Education.

[Tigli et al., 2009] Tigli, J.-Y., Lavirotte, S., Rey, G., Hourdin, V., and Riveill, M. (2009). Lightweight Service Oriented Architecture for Pervasive Computing. *Int. J. of Computer Sci. Issues*, 4(1).

[Triboulot et al., 2014] Triboulot, C., Trouilhet, S., Arcangeli, J.-P., and Robert, F. (2014). Composition et recomposition opportunistes : motivations et exigences. *Journées francophones Mobilité et Ubiquité (UBIMOB), Sophia Antipolis*.

[Triboulot et al., 2015] Triboulot, C., Trouilhet, S., Arcangeli, J.-P., and Robert, F. (2015). Opportunistic software composition : benefits and requirements. In Lorenz, P. and Maciaszek, L. A., editors, *Int. Conf. on Software Engineering and Applications (ICSOFT-EA)*, pages 426–431. INSTICC.

[Vogel, 2018] Vogel, T. (2018). *Model-Driven Engineering of Self-Adaptive Software*. PhD thesis, University of Potsdam, Germany.

[W3C, 2004] W3C (2004). Owl-s : Semantic markup for web services. `https://www.w3.org/Submission/OWL-S/`. Online : accessed 20/10/2020.

[W3C, 2007] W3C (2007). Web services description language. `https://www.w3.org/TR/wsdl/`. Online : accessed 20/10/2020.

[Wang and Vassileva, 2007] Wang, Y. and Vassileva, J. (2007). A review on trust and reputation for web service selection. In *27th international Conference on distributed computing systems workshops (ICDCSW'07)*, pages 25–25. IEEE.

[Wooldridge, 2009] Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition.

[Xiao et al., 2011] Xiao, H., Zou, Y., Tang, R., Ng, J., and Nigul, L. (2011). Ontology-driven service composition for end-users. *Service Oriented Computing and Applications*, 5(3) :159.

[Younes et al., 2019a] Younes, W., Adreit, F., Trouilhet, S., and Arcangeli, J.-P. (2019a). Apprentissage en ligne par renforcement centré utilisateur pour la composition émergente d'applications ambiantes. In *Conférence Nationale d'Intelligence Artificielle (CNIA 2019)*, pages 179–186, http ://www.afia.asso.fr/. AFIA : Association Francaise d'Intelligence Artificielle. `https://afia.asso.fr/wp-content/uploads/2019/12/CNIA_2019.pdf`.

[Younes et al., 2018] Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2018). Towards an intelligent user-oriented middleware for opportunistic composition of services in ambient spaces. In *Proc. of the 5$^{th}$ Workshop on Middleware and Applications for the Internet of Things (M4IoT'18)*, pages 25–30, New York, NY, USA. ACM. `http://doi.acm.org/10.1145/3286719.3286725`.

[Younes et al., 2019b] Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2019b). Principles of user-centered online reinforcement learning for the emergence of service compositions. Technical report IRIT/RR–2019–05–FR, IRIT. https://www.irit.fr/~Sylvie.Trouilhet/publiInfo/IRIT_RR_2019_05_FR.pdf.

[Younes et al., 2020] Younes, W., Trouilhet, S., Adreit, F., and Arcangeli, J.-P. (2020). Agent-mediated application emergence through reinforcement learning from user feedback. In *29th IEEE International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE 2020)*. IEEE Press. https://hal.archives-ouvertes.fr/hal-02895011.

[Zeng et al., 2003] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., and Sheng, Q. Z. (2003). Quality driven web services composition. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, page 411–421, New York, NY, USA. Association for Computing Machinery.

[Zeng et al., 2004] Zeng, L., Benatallah, B., Ngu, A. H., Dumas, M., Kalagnanam, J., and Chang, H. (2004). Qos-aware middleware for web services composition. *IEEE Transactions on software engineering*, 30(5) :311–327.

# Liste des Figures

# Liste des Tables