Master Recherche STIC

—

Réseaux et Systèmes Distribués

Ingénieur « Sciences Informatiques »

—

Systèmes et Applications Réparties

Rapport de Stage

Vérification de l'adaptation dynamique des assemblages de composants

Sébastien BIANCO

27 mars — 15 septembre 2006

Encadrantes :
Anne–Marie Pinna–Dery
Mireille Blay–Fornarino

Laboratoire d'accueil :
I3S — **I**nformatique, **S**ignaux et **S**ystèmes de **S**ophia Antipolis
2000, route des Lucioles
Les Algorithmes - Bât. Euclide B
B.P. 121 06903 Sophia Antipolis - Cedex

Remerciements

Je tiens à remercier Anne–Marie Pinna–Dery et Mireille Blay–Fornarino pour m'avoir proposé un sujet de stage intéressant ; ainsi que les membres de l'équipe RAINBOW pour leur accueil.

Je remercie en particulier Michel Riveill pour nous avoir fourni une salle et un salaire, ainsi que Jean–Yves Tigli et Stéphane Lavirotte pour m'avoir permis de faire évoluer la plateforme Wcomp.

Je dois aussi citer dans cette partie Daniel Cheung pour son aide, Philippe Collet qui a pris le temps de répondre à mes questions sur Fractal, Audrey Oscello qui m'a initié au modèle à composants, ainsi que Vincent Léon qui a accepté, parfois malgré lui, d'écouter mes états d'âmes sur mon sujet lorsque j'étais en difficulté.

Enfin, je dois remercier tendrement Ketty Nguyen qui a eu la patience de relire ce document pour s'assurer qu'il était relativement compréhensible pour les non initiés.

Table des matières

Table des figures

Prélude

Découvrir le monde de la recherche, ainsi que de valider mes cursus d'Ingénieurs et de MASTER Recherche sont les deux buts qui m'ont poussé à faire mon stage dans un laboratoire public.

L'équipe RAINBOW, qui m'a accueilli, m'a proposé un sujet basé sur les modèles à composants avec pour but d'enrichir les plateformes existantes afin d'ajouter des contraintes de comportements. Ces travaux s'inscrivent en marge d'un projet national débutant : le projet FAROS.

Mon rapport sera organisé comme tel. Après avoir brièvement précisé les objectifs de mon stage, je ferais un rapide état de l'art afin d'introduire les notions qui m'ont été utile. Puis, je présenterais ma solution, ainsi que quelques détails sur son implémentation ; avant de conclure sur les perspectives qu'offrent mes travaux.

Je vous souhaite une bonne et, je l'espère, enrichissante lecture...

Chapitre 1

Introduction et objectifs de ce stage

Le monde de l'informatique est un secteur particulièrement bouillonnant où les applications doivent pouvoir évoluer vite et à moindre frais. De plus, l'intégration de l'informatique dans le monde quotidien est grandissante, le monde se dirigeant lentement mais sûrement vers le tout-informatique. Les périphériques se multiplient, et chacuns offrent des services spécialisés. L'un fait de la photo, l'autre lit de la musique, le troisième enfin permet de téléphoner. Les applications naissantes ont souvent pour but de se servir de ses différentes fonctionnalités afin de fournir un service global. On pourrait prendre comme exemple les téléphones mobiles 3G qui utilisent à la fois la communication, le son et l'image en une application permettant la visiophonie.

Comme nous le montre cet exemple, les applications d'aujourd'hui résultent souvent d'un agglomérat de fonctionnalités communiquant entre-elles. Le problème réside dans l'hétérogénéïté des périphériques. Dans le cas des téléphones mobiles, tous ne possèdent pas de module de lecture de musique, ou d'appareil photo. Il est donc important que les applications qui les utilisent soient capables de s'adapter à leur support, puisqu'il est bien trop coûteux de créer une applications spécifique à chaque terminal.

Ce stage est donc basé sur l'étude de plateformes prévues pour pouvoir se configurer dynamiquement. L'objectifs principal est de permettre un contrôle de l'utilisateur sur cette reconfiguration à partir de spécifications décrite à part du code métier. Ce contrôle doit pouvoir être modulé par l'environnement.

Afin d'illustrer mon propos, voici l'exemple qui va constituer le fil rouge de ce rapport et qui a guidé la recherche de notre solution. Cet exemple est conçu autour de trois scénarii d'utilisation qui illustre cette notion de contrôle.

1.1 L'exemple fil rouge

Le contexte

L'exemple choisi a pour cadre une application domotique. Nous nous plaçons dans un monde idyllique (qui pourrait sembler un enfer pour certains) où tout élément électrique de la maison est relié à un réseau informatique, c'est à dire, par exemple, que si nous nous plaçons dans un monde immergé dans Ethernet, alors on suppose que chaque élément du réseau électrique possède une adresse IP[1] ainsi que le micro-serveur capable de répondre à différentes requêtes (l'analogie idéale est de supposer que ces éléments sont en fait des web services[2]).

[1] Merci IPv6 ;)

[2] Technologie permettant à des applications de dialoguer à distance via Internet indépendamment des plates-formes et des langages sur lesquelles elles reposent. Les services Web s'appuient sur un ensemble de protocoles standardisant les modes d'invocation mutuels de composants applicatifs. La technologie des Services Web est aujourd'hui de plus en plus incontournable et se présente comme le nouveau paradigme des architectures logicielles. Cette technologie englobe de nombreux concepts et tend à s'imposer comme le nouveau standard en terme d'intégration et d'échanges Business–to–Business. (*Dico Du Net – http://www.dicodunet.com/definitions/normes/service-web.htm*)

Les différents éléments de notre exemple (cf figure 1.1) seront :
- des lampes
- des interrupteurs
- des capteurs de présence
- des ordinateurs de poche/P.D.A.[3]

L'interrupteur permet de commander à distances les lampes, tandis que le détecteur de présence réagit lorsqu'une personne est dans son champs de détection.

Le P.D.A. joue, dans notre exemple, le rôle d'un interrupteur mobile. Il faut imaginer son utilisation dans un cadre bien plus large où une application unique permet de gérer l'ensemble des éléments électriques de notre maison. Une telle application semblerait parfaite si elle était capable de détecter automatiquement les nouveaux composants arrivant dans notre réseau (détection de nouveaux services), les composer pour en faire des entités de plus haut niveau et même adapter son interface graphique. Cependant, ce sujet étant bien trop ambitieux pour notre stage, je ne me suis pas occupé de développer un tel programme.

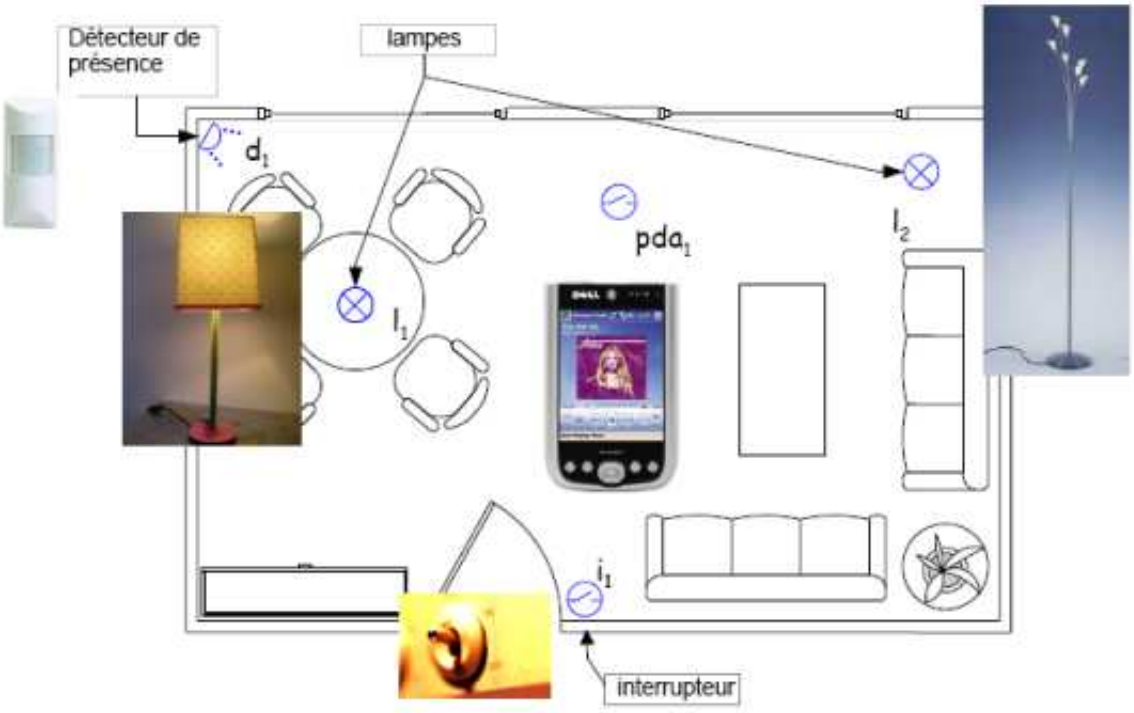

Fig. 1.1 – Exemple fil rouge : Une application domotique pour gérer l'éclairage de lampes

De nombreux travaux de recherche sont cependant en cours pour permettre la réalisation de services complexes qui se reconfigureraient automatiquement en fonction de l'environnement. Dans cette optique, les travaux de Sonia Ben Mokhtar à l'INRIA Rocquencourt [4] proposent un protocole de découverte de services, ainsi qu'un algorithme permettant d'agréger des services répartis afin d'obtenir une application possédant la sémantique spécifiée par l'utilisateur. Ceux de Clémentine Nemo au laboratoire I3S ([18] ou [17]) sont eux plus orientés sur la composition d'orchestration de service. On peut également trouver des travaux sur la composition de GUI[4] ([19] ou [20])pour permettre une intégration rapide des services au sein d'une application

[3]Personal Digital Assistant (en français, Assistant Personnel)
[4]Graphical User Interface

utilisateur.

Mes travaux sont donc périphériques à de nombreux autres, et s'inscrivent dans la mouvance de l'émergence de plateformes à composants dynamiquement reconfigurables, en se focalisant sur la mise en place de contrats d'assemblage adéquats à ce type d'application.

Définition des cas d'utilisations

Les plateformes à base de composants proposent une conception et programmation des applications par assemblage. L'intérêt principal est la (re)configuration possible "à chaud" et automatique en fonction de paramètres fixés par l'utilisateur. Cette puissance est toute indiquée dans des applications proches de celles que nous envisageons. Cependant, il est nécessaire de poser des jalons afin que l'application reste cohérente et qu'elle réponde aux besoins fixés par l'utilisateur.

C'est avec cette envie de contrôle de l'utilisateur sur la machine[5] que nous avons entamé le stage. Cependant, nous souhaitions nous restreindre à des cas de contrôles simples mais inabordés.

Pour mettre en œuvre le contexte décrit ci-dessus, il nous fallait trouver une plateforme permettant de mélanger des composants physiques (nos lampes, interrupteurs, détecteurs de présence) avec des composants logiciels (le programme qui va simuler l'interrupteur sur notre PDA, i.e. le programme qui va permettre de commander l'éclairage des lampes).

Cette étape franchie, nous voulions pouvoir placer des contraintes d'utilisations sur notre application. Pour cela, nous avons imaginé trois scénarii d'utilisations :

- Nous souhaitons s'assurer que les lampes l_1 et l_2 seront bien synchronisées. C'est à dire s'assurer que les lampes s'allument et s'éteignent conjointement au signal de l'interrupteur.
- Nous souhaitons que lorsque le PDA est présent dans le système, ce soit lui qui contrôle les lampes et non pas l'interrupteur mural.
- Nous souhaitons n'allumer qu'une lampe en fonction de la présence ou non d'une personne dans le champ du détecteur (i.e. Allumer l_1 si le détecteur indique la présence d'une personne, l_2 sinon).

Ces contraintes pourraient être exprimées par l'utilisateur indépendamment de l'application principale et être vues comme des contraintes liées à de la qualité de service. Il serait alors bienvenu de comparer ces descriptions à des "Contrats".

1.2 Définition des objectifs

À partir de cet exemple, les besoins primaires de notre application sont :

- une plateforme autorisant une configuration et reconfiguration dynamique autour des différentes fonctionnalités,
- une notion de contrat permettant de contrôler cette configuration,
- un système de réactions aux changements de l'environnement (entrée/sortie d'une fonctionnalité) ou à un événements survenant au cours de l'application.

[5] Matrix doit rester de la Science-Fiction

Ainsi, nous pouvons reformuler nos objectifs de la manière suivante. Nous voulons :
- Contrôler l'évolution dynamique d'une application en fonction du contexte.
- Fournir les outils pour exprimer les contrôles sur l'évolution. Nous nous appuierons plus précisément sur le concept de contrat. Cette notion n'étant pas parfaitement définie, nous pourrons en introduire différents types.
- Permettre l'évolution du contrôle de l'application en cours d'execution : de nouveaux contrats pourront-être définie par l'utilisateur, ou par le système lui même afin de résoudre des conflits survenant durant le runtime (une validation de l'utilisateur étant sûrement nécessaire).
- Exprimer les contrats séparément les uns des autres. Ils s'agit alors de fournir les moyens de les composer et de choisir ceux qui seront actifs ; ainsi que de gérer les éventuelles incohérences.

Au vu de ces besoins et objectifs, il est important de se focaliser tout particulièrement sur les différentes plateformes à composants existantes. Nous retiendrons celles qui pourraient prendre en compte les notions qui nous intéressent, à savoir la gestion de contrats et de composants matériels.

Ensuite, il sera opportun d'évaluer les manques de ces plateformes et de proposer une solution répondant aux problématiques que nous nous sommes fixées.

Enfin, il serait de bon aloi d'étudier les perspectives qu'offre la solution proposée ainsi que les nouvelles problématiques soulevées.

Chapitre 2

État de l'art

2.1 Les modèles à composants

Depuis plusieurs années la communauté scientifique[1] poursuit des objectifs visant à faciliter la construction économiquement compétitive de logiciels. Les travaux entrepris visent à rendre ces logiciels plus fiables, plus rapides à développer et à maintenir, plus adaptables et réutilisables par morceaux. Ainsi les langages informatiques ont évolué pour s'architecturer autour de la notion de classe.

Le niveau d'abstraction suivant est la notion de composant logiciel. On peut voir le composant logiciel comme une boite noire offrant un service donnée. Cette boite possède un certain nombre d'entrées à partir desquelles on obtient un "service" et requiert d'autres composants qui doivent être connectés sur ses ports de sorties. Cette notion est à rapprocher de celle de la "puce" au sens de l'électronique. Avec cette vision, construire une application revient à faire un "câblage" entre des composants déjà développés (souvent appelés "Composants sur Étagères").

De tels modèles bouleversent les processus du génie logiciel traditionnel, en mélangeant les propriétés statiques et dynamiques, les propriétés fonctionnelles et celles concernant la qualité de service. Pour maîtriser le fiabilité d'applications à base de tels composants, il est nécessaire d'intégrer les vérifications dans des processus de développement, à base de (re)configuration dynamique et de négociation.

Aujourd'hui les composants intègrent peu à peu le monde de l'entreprise et les processus de développement. Les modèles à composants les plus connus et les plus utilisés sont CCM [2], EJB [3] ou encore COM [1] de Microsoft.

Cependant de nombreuses plateformes à composants sont en cours de développement dans les laboratoires de recherche du monde entier.

Dans les prochaines sections je vais présenter celles auxquelles je me suis particulièrement intéressé au cours de mon stage : Wcomp puisqu'elle possède pour particularité de travailler avec de composants mixtes, c'est à dire des composants à la fois hardware et software, particulièrement proches de l'application domotique qui constitue mon exemple[2], et ConFract qui introduit la notion de contrat entre composants. Je présenterai également le modèle d'interaction basé sur le langage ISL et son implémentation NOAH ; ainsi qu'une nouvelle technologie qui m'a été utile dans l'implémentation de ma solution : UPnP.

[1] cf Manifeste du groupe Objects, Composants et Modèles (O.C.M.) (`http://www-valoria.univ-ubs.fr/Jacques.Malenfant/ALP.OCM/manifeste.html`) ainsi que l'introduction de l'article de Philippe Collet à ConFract [13]

[2] Remarque : mon exemple de travail est librement inspiré de celui des articles de Daniel Cheung [10] et [11] qui travaille lui aussi sur cette plateforme Wcomp

2.2 ISL – *Interaction Specification Language* et son implémentation NOAH

"The architecture of a software system defines that system in terms of components and interactions among those components" [21]

Comme nous l'avons évoqué dans la partie 2.1, un programme à composants peut-être vu comme un assemblage de boîtes noires. C'est la description de cette assemblage qui est important. Ainsi les travaux menés par l'équipe RAINBOW[3] du laboratoire I3S ont abouti sur l'élaboration d'un langage permettant de décrire les interactions entre les composants : ISL [5].

Dans les paragraphes qui vont suivre, je vais commencer par énoncer les principes et les avantages de l'utilisation d'un modèle d'interactions, puis j'évoquerai rapidement les apports du service d'interaction, enfin j'appliquerai ce modèle à notre exemple ; ce d'après [6] et [9].

2.2.1 Intérêt d'un modèle d'interaction

Le but d'un service d'interaction est de permettre l'adaptation dynamique d'applications à base de composants. Ce service est basé sur un modèle d'interaction.

Le principe de base est de décrire dans un *Pattern* un ensemble de règles d'interaction. Ces règles sont enregistrées sur un serveur puis instanciées au besoin sur les instances de composants. Ainsi, cette méthode permet de créer dynamiquement des liens entre les composants, et donc de modifier le comportement de l'application en fonction de l'environnement. On remarquera que les *Pattern* d'interaction sont l'abstraction du concept d'interaction, et que donc, la différence entre un *Pattern* et une interaction est du même ordre que celle entre une classe et un objet.

Les principes de base du modèle d'interaction sont :
- Éviter les inconsistences qui peuvent découler d'un groupe d'adaptation
- Diriger la composition des interactions automatiquement
- Assurer l'interopérabilité des interactions entre des composants hétérogènes
- Permettre une communication directe entre les composants en interaction, sans passer par un serveur qui centraliserait la gestion des interactions

Le langage ISL, qui permet de mettre en oeuvre ce modèle d'interaction, est utilisé pour spécifier les liens entre les composants indépendamment de tout langage de programmation. Les éléments de base sont les opérateurs de condition (if, then, else, endif), l'opérateur de séquence (;), l'opérateur de parallélisation(//), l'opérateur d'affectation (:=), les opérateurs de gestion des exceptions (try, catch) ou encore l'opérateur * qui permet de signifier par exemple que tout les signaux sortant d'un composants sont concernés par la règle.

Le langage ISL a été développé pour permettre la composition automatique de plusieurs schémas concernant un même composant. C'est pourquoi il a été défini des opérateurs destinés à cette opération, afin de permettre une composition déterministe. Ces opérateurs ISL sont *call* et *delegate*. Ils sont utilisés lors d'une redéfinition d'un port. *Call* exprime que l'action définie dans un autre schéma peut apparaître si un conflit survient lors de la composition. Au contraire le mot-clé *delegate* suggère un remplacement par la définition de l'autre schéma.

En parallèle, il existe un nombre de règles qui permet au serveur qui va gérer ces patterns, de les mélanger afin de générer un assemblage cohérent. Ce processus de composition est associatif et commutatif, mais peut cependant échouer, par exemple dans le cas où la composition est non déterministe. Malgré tout, le mécanisme de composition est essentiel pour maintenir la cohérence globale et la consistance de toute les interactions.

2.2.2 Mise en oeuvre du service d'interaction NOAH

Le point d'orgue de ce modèle est la mise en place d'un service d'interactions. Ce service permet le création d'interaction entre composants hétérogènes. L'implémentation qui est actuellement faite de ce modèle

[3] `http://rainbow.essi.fr/`

s'appelle **Noah**[4].

Ce service est basé sur la complémentarité entre un serveur d'interactions qui va gérer les pattern d'interactions et offrir des méthodes pour naviguer dans le graphe d'interactions ; et des composants préparés pour diriger et interpréter les interactions. Le comportement de ces composants est dynamiquement modifié lors de l'ajout ou de la suppression d'une interaction.

L'implémentation actuelle du serveur d'interaction de NOAH est basée sur Java RMI, mais celui-ci peut-être aussi interrogé par le biais d'un service web, afin de pouvoir être accessible par les composants .NET. Ce serveur est donc chargé de gérer le cycle de vie de ces interactions, à savoir : les enregistrer, les instancier et les détruire !

Les composants qui peuvent interagir obéissent au trois règles suivantes :

- Ils peuvent sauter du fil principal d'exécution, à un contrôleur local lorsqu'ils reçoivent un signal de notification
- Ils peuvent mêler et démêler dynamiquement des règles d'interactions
- Ils peuvent adresser directement un message à un autre composant, avec lequel il est connecté, sans passer par le serveur d'interactions.

Pour cela, des utilitaires ont été développés pour transformer des composants normaux en composants qui peuvent interagir.

Les communications entre les composants hétérogènes sont rendues possibles par l'utilisation du XML.

2.2.3 Notre exemple fil rouge en application

Reprenons maintenant notre exemple et essayons maintenant d'y incorporer de l'ISL.

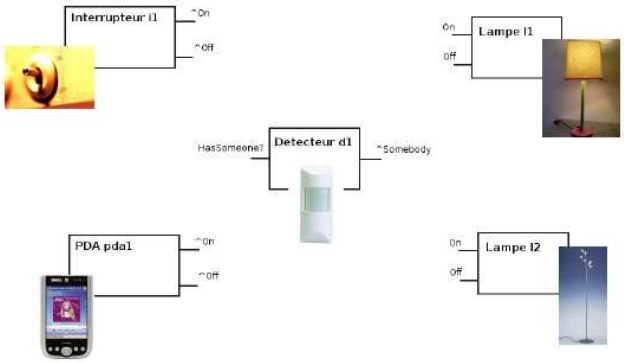

Fig. 2.1 – Exemple fil rouge formalisé pour une plateforme à composants

Comme nous le montre la figure 2.1, l'exemple se modélise donc sous l'apparence de 5 composants représentés chacun par des boîtes avec leurs entrées et sorties. Les lampes l_1 et l_2 ont deux entrées afin de recevoir les ordres allumer/éteindre. Les interrupteurs i_1 et pda_1 ont deux sorties afin d'envoyer ces ordres. Enfin, le détecteur de présence possède une entrée permettant de l'interroger sur la présence d'une personne dans le champ de detection, et une sortie déclenchée par l'arrivée d'une personne dans cette zone.

Utilisons maintenant le langage de spécification de liaison ISL pour décrire les interactions entre l'interrupteurs et nos lampes.

[4]New Object Adaptation Hardware – `http://rainbow.essi.fr/noah/`

Description des interactions entre un interrupteur et une lampes

```
interaction int_lam_on (Lampe l, Interrupteur i){
    i.on() -> i.call // l.on()
}

interaction int_lam_off (Lampe l, Interrupteur i){
    i.off() -> i.call  //  l.off()
}
```

2.2.4 Les apports d'ISL

Comme nous l'avons vu au cours de cette partie, les apports d'ISL sont multiples. Le point cependant qui nous intéresse est la possibilité qu'offre ISL pour la configuration dynamique en fonction des objets instanciés et des règles d'interactions proposée par l'utilisateur.

La possibilité de lier des composants hétérogènes en décrivant les liaisons dans un langage indépendant de toute implementation est également une plus-value appréciable.

2.3 Wcomp

Toute la partie théorique sur Wcomp est rédigée à partir des documents de Daniel Cheung Foo Wo [10] et [11].

Wcomp est un projet qui a pour but de mettre en place une plate-forme de développement rapide pour les systèmes d'informatique ambiante (système multi-dispositifs). Ce projet a abouti à un environnement de développement à base de composants logiciels. Un composant est une instance de classe (Java ou C# selon l'implémentation). Il est doté d'entrées et de sorties ; typiquement des **interfaces serveur** qui sont des ensembles d'opérations et des **interfaces client** qui se composent d'évènements qui peuvent être émis.

Les composants Wcomp se découpent en deux catégories : les composants logiciels (appelés simplement **composants**) et les **composants mixtes** c'est à dire des composants logiciels représentant un équipement matériel.

La communication entre les composants est de type asynchrone. Les composants émettent des événements via les interfaces client. D'autres composants sont mis en écoute de ces évènements. Ils sont notifiés via leur interface serveur par une transformation adéquate de l'événement en un envoi de message. La transformation est prise en compte par le composant émetteur. Dans un assemblage, on différencie les composants dits actifs des composants passifs. Les composants Wcomp sont qualifiés de **passifs**, s'ils ne créent pas de nouveaux fils d'exécution (thread) ou de tâches, et **actifs** dans le cas contraire. On peut noter qu'il est possible de rendre actif un composant passif (en l'associant avec un composant actif) ou de passifier un composant actif (grâce à un composant Wcomp de bufferisation).

Wcomp fonctionne sur un découpage **Designer**/**Container**. Le *Container* a en charge la gestion des composants qui forment l'application finale. C'est lui qui met en place les connections entre les différents briques applicatives. Le *Designer* est une application qui communique avec le *Container* afin de le diriger. C'est par le *Designer* que l'on ordonne au *Container* d'ajouter/retirer des composants. C'est aussi le *Designer* qui indique les liaisons à créer/détruire.

2.3.1 ISL4WCOMP

Afin d'automatiser la création de liaisons entre les composants, il a été créé un nouveau *Designer* qui utilise un langage de description de ces interactions. Je vais donc faire le point sur ce langage, en particulier sur ses similitudes avec le langage ISL mentionné dans la partie 2.2 ; puis j'évoquerai rapidement son mode de fonctionnement.

ISL vs ISL4WCOMP

ISL4Wcomp est un langage fortement inspiré du langage ISL, mais spécialisé pour la plateforme Wcomp. Il en découle donc un certain nombre de différences.

Si en ISL c'est le typage qui joue un rôle fondamental, dans ISL4Wcomp c'est le couple Type/Nom qui est important. Ainsi, si une description ISL4Wcomp mentionne une liaison entre le composant *comp1* de type *A* et le composant *comp2* de type *B*, alors le *Designer* ne réagira pas à la présence du couple *comp1* de type *A* et *comp3* de type *B*. C'est pour cela que je vais appeler désormais ces descriptions **Interaction Potentielle**.

Au niveau du modèle, à l'origine, le langage ISL permet de redéfinir les appels de méthodes par une redirection de cet appel vers un morceau de code chargé d'appeler la véritable méthode au moment opportun. Le modèle Wcomp utilise bien des appels de méthodes en entrée de ses composants (ceux ci peuvent donc bien être réécrits en ISL) mais utilise aussi des événements en sortie. C'est pourquoi ISL a du être modifié pour pouvoir gérer ces ports de sortie événementiels.

Pour marquer l'évolution,la syntaxe de decription des interactions potentielles a donc été homogénéisée et s'est inspirée de langage haut niveau tel que C# ou Java. Ce nouveau langage de construction est appelé **ISL4Wcomp**. Le tableau 2.1 décrit les codes essentiels de ce langage.

Symbole	***Description***
∧	Exprime l'émission d'un événement
:-	Explicite la réécriture du point de jonction
_call	Désigne le comportement initial
;	Exprime une séquence
//	Exprime un ordre indéterminé

Tab. 2.1 – Tableau récapitulatifs des commandes de base d'ISL4Wcomp

Architecture d'ISL4WComp

L'architecture de ISL4WComp est différente de celle d'ISL. En effet, on n'utilise pas le serveur d'interactions pour gérer le cycle de vie des interactions. À la place, un ensemble d'outil a été développé pour composer les interactions, pour appliquer les interactions aux instances de composants et pour transformer les différentes interactions en assemblage de composants.

Les composants peuvent être utilisés dans plusieurs schémas d'interactions. Il faut alors les composer. Un premier outil (*ISLComposer*) réalise cette tâche. Le schéma composé est ensuite communiqué à un second outil (*ApplyISL*) qui va mettre en correspondance les variables définies et les instances d'objets. Enfin, le dernier outil (*ISLTranslator*) traduit ce modèle appliqué en une liste de commandes permettant de créer et de lier les composants (commandes *add_ component*, *remove_ component*, *link*,*unlink*...). La figure 2.2 illustre ces différentes étapes, menant du contrat ISL à l'assemblage de composant.

2.3.2 Retour sur l'exemple fil rouge

Lors de la partie 1.1, j'ai introduit un problème de domotique élémentaire. Il est maintenant temps de le revoir à la lumière de modèles à composants et en particulier de le formaliser sur notre plateforme Wcomp, consolidée par ISL4Wcomp.

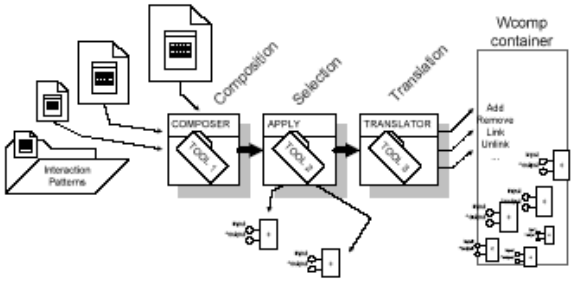

Fig. 2.2 – Mise en œuvre de ISL4WComp

La figure 2.1 nous montre comment peut-être modélisé notre problème sur la plateforme Wcomp. Décrivons les interactions entre ces composants avec le language ISL4Wcomp que nous venons d'étudier. Nous avons donc deux séries d'interactions.

Description des interactions potentielles entre entre l'interrupteur et les lampes

```
Schema int_lam1 (Lampe l1, Interrupteur i1){
    i1.^on()  { call  || l1.on() }

    i1.^off() { call  ||  l1.off() }
}

Schema int_lam2 (Lampe l2, Interrupteur i1){
    i1.^on()  { call  || l2.on() }

    i1.^off() { call  ||  l2.off() }
}
```

Description des interactions potentielles entre entre le pda et les lampes

```
Schema pda_lam1 (Lampe l1, PDA pda1){
    i1.^on()  { call  || l1.on() }

    i1.^off() { call  ||  l1.off() }
}

Schema pda_lam2 (Lampe l2, PDA pda1){
    i1.^on()  { call  || l2.on() }

    i1.^off() { call  ||  l2.off() }
}
```

Si l'on n'impose aucune limitation, et qu'à un instant donné l'ensemble des composants pda_1, l_1, l_2, i_1 sont présents au sein de notre *Container*, alors le *Designer* va créer, après composition des schémas, un assemblage ressemblant à la figure 2.3

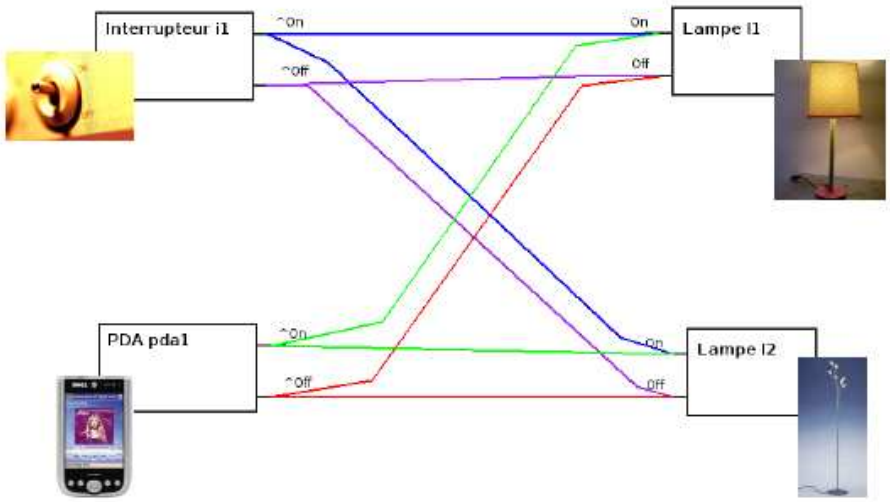

Fig. 2.3 – Exemple fil rouge : l'assemblage réalisé en dehors de toutes contraintes

2.3.3 Conclusion sur Wcomp

Les avantages que nous apporte Wcomp sont significatifs. Cette plateforme, qui gère de manière transparente des composants mixtes, est indiquée dans notre application domotique.

De plus le découpage *Container*/*Designer* nous offre beaucoup de souplesse puisque un même *Container* peut travailler avec différents *Designer* et il devient alors possible de créer de multiples interfaces (à prendre ici au sens Interface Homme Machine) permettant de piloter et de reconfigurer si besoin une même application.

Enfin, l'utilisation du *Designer* ISL4Wcomp permet d'automatiser la création d'un assemblage (et donc, d'une application), en spécifiant seulement les interactions possibles entre les composants (et ce indépendamment de la future application générée).

2.4 ConFract

Je vais présenter les travaux effectués au niveau de la plateforme Fractal, et de sa version évoluée pour la gestion de contrat ConFract. Pour que ce paragraphe soit le plus complet possible, je commencerai par faire une courte présentation du modèle à composants Fractal, avant de me focaliser sur le langage de contractualisation CCL-J utilisé par Confract et son intégration dans le système Fractal.

2.4.1 Fractal, un modèle à composants

Cette partie théorique sur la plateforme est écrite d'après le rapport de stage d'Hervé Chang [8] effectué au laboratoire I3S en 2003.

Fractal est un modèle de composants développé depuis Janvier 2002 par France Télécom R&D au sein du consortium Objectweb[5], qui réunit des organisations comme Bull, l'INRIA et France Télécom R&D.

Fractal est à la fois un framework général de conception mais aussi une plate-forme opérationnelle avec une API de manipulation des composants. Le but du modèle Fractal est de définir un modèle de composants logiciels général, modulaire et extensible fondé sur une base rigoureuse [7]. Il se distingue des modèles de composants actuels par :

[5]ObjectWeb Consortium, `http://www.objectweb.org/` 1999.

- son caractère plus abstrait et générique : il reste indépendant de toute implémentation et présente l'avantage de pouvoir servir de base à diverses extensions ;
- une expressivité élaborée : il permet la composition hiérarchique des composants (un assemblage de composants peut être lui-même considéré comme un composant) ainsi que le partage de composants ;
- des capacités d'adaptation et de reconfiguration dynamique particulièrement développées ;
- la capacité pour un composant d'exprimer clairement, au travers d'interfaces fournies et requises, les services qu'il apporte et ceux qui lui sont nécessaires pour réaliser ses propres services.

Le modèle à composants fractal peut être approché selon deux niveaux. Le niveau abstrait et le niveau concret. Le niveau concret (plus proche de l'implémentation) enrichit les concepts du modèle abstraits.

Le modèle abstrait de Fractal

Trois concepts majeurs sont définis dans la modélisation abstraite de fractal : le composant, le contrôleur et le signal. Les concepts de composant et de signal sont fondamentaux dans le milieu des plateformes à composants.

Un **composant** est une boîte noire offrant un certain nombre de services. Ce composant présente une série d'interfaces permettant à d'autres composants d'utiliser les services offerts (Interface fournie), une autre série qui doit être raccordée à des composants offrant des services qui va permettre à notre composant de fonctionner correctement (Interface Requise).

La particularité de Fractal est d'offrir une dernière série d'interfaces qui permettent de piloter le composant, et de gérer sa hiérarchisation. Un composant Fractal peut-être **primitif** (il encapsule directement du code exécutable) ou **composite** (il encapsule alors un nombre de composants à des niveaux d'emboîtement quelconque) et peut être **partagé** pour la modélisation de ressources (connexions, pools, caches,...) ou des activités (transactions, workflows,...).

La figure 2.4 présente l'organisation logique d'un composant Fractal en contrôleur et contenu. Le contrôleur intercepte l'ensemble des informations transitant sur les interfaces et gère le fonctionnement du composant selon les informations qui lui proviennent par les interfaces de contrôle.

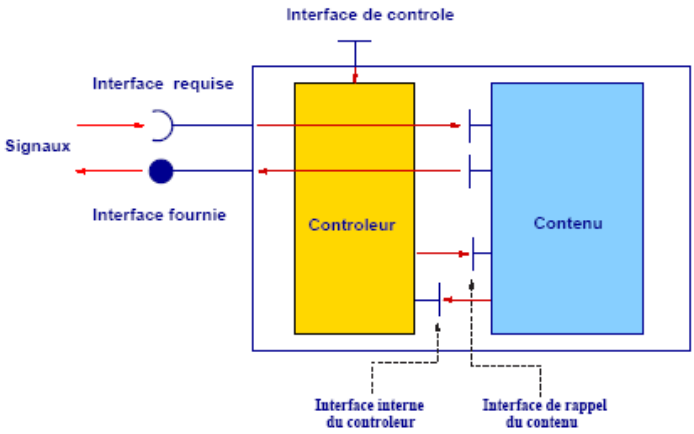

Fig. 2.4 – Organisation logique d'un composant en contrôleur et contenu

Un **contrôleur** correspond à une entité qui permet de fournir des capacités d'introspection du contenu du composant. Il applique le principe de séparation des préoccupations à la gestion et à la surveillance de chaque composant en inspectant les signaux entrants et sortants du composant (approche Orienté Aspect [14]). C'est au contrôleur que revient la gestion des aspects d'administration (reconfiguration dynamique) et

l'intégration des aspects non fonctionnels (transactions, sécurité). Il peut aussi modifier le comportement des sous-contrôleurs dans le cas d'une hiérarchie.

Un **signal** est un message entrant ou sortant d'un composant, typiquement les invocations d'opérations entrantes et sortantes. L'interception, par le contrôleur de ces signaux, permet de connaître l'état courant du composant et sert notamment à intégrer les aspects non fonctionnels.

Le modèle concret de Fractal

Le modèle concret de Fractal raffine le modèle abstrait en y ajoutant de nouvelles précisions concernant les liaisons. Il précise les notions d'interface et introduit celles de liaison, d'instanciation et de noms.

Une **interface** représente le point d'accès aux fonctionnalités d'un composant. Pour un composant, on distingue ici clairement les interfaces **fournies** (les services publiés que propose le composant), **requises**(les services dont il a besoin pour pouvoir réaliser ses services) et de **contrôle** (les services permettant de gérer le composant). Notez que lors de l'implémentation des composants, les interfaces sont notées comme **obligatoires** ou **optionnelles** selon leur degré de nécessité dans le programme. On distingue aussi les interfaces internes, qui exposent les services à l'environnement internes du composant, des interfaces externes, qui exposent les services à l'environnement externe du composant. Pour les composites, il existe un couplage fort entre les interfaces internes et externes puisque l'existence d'une interface interne (resp. externe) implique l'existence d'au moins une interface externe (resp. interne).

Une **référence d'interface** est un nom qui désigne une interface et qui a le type de l'interface désignée.

Les **liaisons** représentent les interactions entre les composants. Elles peuvent être simples en reliant exactement deux interfaces Fractal, ou composées en reliant plusieurs composants.

L'**instanciation** de composant est possible selon deux mécanismes : par l'utilisation d'une fabrique (Factory) ou par l'instanciation à partir d'un modèle (Template).

Un **nom** de composant permet de le retrouver de manière univoque.

Notre exemple fil rouge en fractal

Reprenons notre exemple d'application domotique. D'un point de vue du formalise les différences avec Wcomp sont notables.

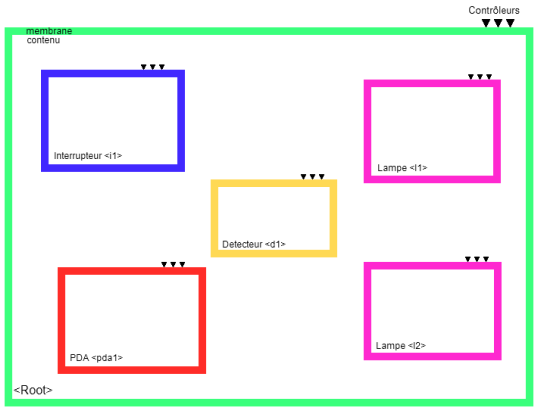

Fig. 2.5 – Représentation de notre exemple selon le modèle abstrait de Fractal

Tout d'abord il est nécessaire de placer nos composants dans un composite <root> qui sera chargé de gérer l'assemblage des composants qu'il contient.

Ensuite, il va être nécessaire de définir les Interfaces qui vont relier les composants. Les composants pourront être connectés sur un port que s'ils ont la même interface.

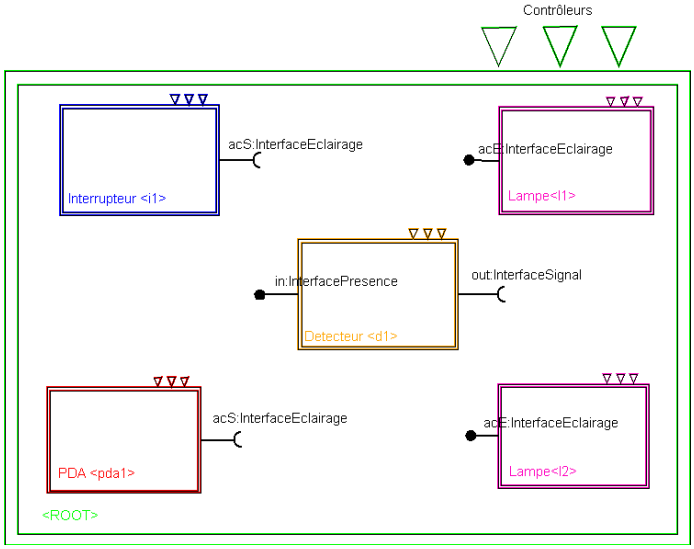

Fig. 2.6 – Représentation de notre exemple selon le modèle concret de Fractal, sans liaisons entre les composants

Enfin, les liaisons entre les différents composants doivent être programmées, même si un langage de description de l'architecture (ADL) est en cours de réalisation. Il est cependant possible de modifier l'assemblage en temps réel, mais cela n'est pas fait automatiquement contrairement à Wcomp assorti du *Designer*.

Ces deux premières particularités sont mises en oeuvre sur les figures 2.5 et 2.6. Les interfaces de liaisons sont décrites ci-dessous. L'implementation actuelle de fractal étant en Java, les composants le sont aussi (et donc la description de leur interface).

Remarque : L'exemple a été un peu enrichi au niveau de l'interface d'éclairage pour pouvoir montrer dans la partie 2.4.2 l'intérêt de ConFract.

```
public interface IntefaceEclairage {
    public void on();
    public void off();
    public boolean isSwitchedOn();
    public int timeOfUse()
}

public interface InterfaceSignal {
    public void somebody();
}

public interface InterfacePresence{
    public boolean hasSomeone();
}
```

2.4.2 Confract, une extension à fractal pour gérer des contrats

Cette sous-section est écrite à partir du rapport d'Hervé Chang [8] précédemment cité, ainsi que de celui de Benoît Vallette d'Osia qui a effectué son stage au sein du laboratoire I3S en 2005 [23].

La notion de contrat en programmation, afin de rendre les programmes plus robuste et exempt de bogues, a été introduite dans les années 1994 par Bertrand Meyer et le langage Eiffel [16]. Ce langage introduisait entre autre les notions de pré-condition et de post-condition sur les fonctions ainsi que d'autres structures de contrôle (variant et invariant pour assurer la terminaison de boucle). Ces principes ont d'ailleurs été repris pour les appliquer au langage Java avec les travaux portant sur JML[6] [22].

Les contrats constituent un moyen pour faciliter la programmation par assemblage et garantir la qualité du système. En utilisant des conditions qui sont testées en début et à la fin de l'appel des méthodes, ils permettent de déterminer si l'exécution est valide par rapport à ce qu'elle doit faire. De ce fait, les contrats répondent aux besoins de spécification, de vérification et de validation.

Une spécification est un ensemble de propriétés qui décrit des contraintes logiques qui doivent être satisfaites à la configuration ou au cours du cycle d'exécution. Ces propriétés sont définies dans un contexte qui précise la localisation spatiale (le composant) et temporelle (l'instant du workflow, l'appel d'une méthode).

Dans le modèle ConFract, un contrat est défini comme étant un accord entre plusieurs parties qui spécifie les droits et les devoirs de chacune des parties. Il se compose d'une liste de spécifications et d'une liste de participants. On distingue plusieurs types de contrat dans le système ConFract :

- les **contrats d'interfaces** pour les relations point à point. Les spécifications qui sont retenues ne citent que les méthodes et les entités qui sont visibles à ces interfaces.
- les **contrats de composition externe** pour les relations de contenance entre les composants. Ils sont postés sur la membrane externe de la membrane d'un composant et sont formés de spécifications ne citant que des interfaces externes. Ils expriment des règles d'utilisation (pré et postconditions) et de comportement (invariant) par rapport aux autres composants de l'assemblage (donc relatifs à l'interface externe).
- les **contrats de composition interne** pour les relations entre les composants à l'intérieur d'un composite. Ils sont postés sur la face interne de la membrane d'un composite. Ils sont formés de spécifications qui ne citent que des interfaces internes du composant et des interfaces externes de ses sous composants. Ils expriment particulièrement des règles d'assemblage et des comportements par rapport aux autres éléments du composite (donc relatifs à l'interface interne).
- les **contrats de bibliothèque** sont des contrats surtout fonctionnels qui sont définis sur des unités réutilisables du langage sous-jacent, classes ou interfaces dans le cas de Java. Ils ne sont pertinents que pour l'implémentation des composants ou de leurs interfaces et ne participent pas aux négociations d'assemblage.

Les contrats en CCL-J

Les propriétés définissant un contrat sont exprimées selon un formalisme particulier définissant le contexte et les relations logiques. Potentiellement, des formalismes de spécifications relativement différents pourraient être utilisés. Cependant, à ce jour, le système ConFract est uniquement fourni avec le langage d'assertions CCL-J[7], dédié aux composants hiérarchiques. Ce langage, dérivé du langage OCL, repose sur des assertions exécutables, organisées en préconditions, postconditions et invariants qui peuvent être vérifiées lors de la configuration ou de l'exécution des composants.

En CCL-J, les contraintes exprimées sont séparées des descriptions de signatures et le mot-clé **context** permet de préciser le contexte d'application des contraintes. Le contexte peut être une interface (au sens de Java) ou une méthode d'interface.

Le mot-clé **on** permet de désigner le type de composant sur lequel repose les interfaces Fractal désignées.

La notation <**...**> sert à désigner un type de composant, une instance de composant ou le Template d'un composant.

CCL-J définit entre autres les types de spécifications correspondant aux mot-clés suivants : **pre** (pour les préconditions), **post** (pour les postconditions), **inv** (pour les propriétés devant-être vraies durant toute la période d'exécution), **rely** (pour les propriétés censées être vraies durant l'appel d'une méthode).

[6] Java Modeling Language – `http://www.cs.iastate.edu/~leavens/JML/`

[7] *Component Constraint Language for Java*

L'accès à l'attribut X d'un composant doit donc être fait par l'intermédiaire du contrôleur. On trouve aussi l'opérateur **.attributes.** dans certaines documentations (notamment [8] et [12]), qui possède la même sémantique, mais qui n'est pas implémenté.

Enfin, le générateur ***** permet de factoriser l'expression d'une contrainte qui porte sur différents contextes.

Application à notre exemple

Mettons cette syntaxe à l'œuvre pour définir quelques contrats sur notre exemple fil rouge.

Spécifications d'interface

– *Spécification 1* : On souhaite effectuer les actions *on* ou *off* sur les lampes, uniquement si celles-ci ne sont pas dans l'état que l'on cherche à atteindre. En effet, on va supposer que les lampes ont un petit compteur qui s'incrémente à chaque éclairage, à des fins de statistiques. On définit donc un contrat point-à-point entre l'interface *acS* de *Interrupteur* et l'interface *acE* de *Lampe*.

```
context  void acS.on()
    // precondition : la lampe n'est pas déjà éclairée.
    pre : !acE.isSwitchedOn()
```

Spécifications de composition

Remarque : Notre exemple n'est pas suffisamment complexe pour nécessiter l'utilisation de composite, autre que le composite de base *ROOT*. C'est pour cette raison que je ne ferais pas la distinction ici entre contrat de composition interne et externe. Les contrats que j'énoncerais appartiennent plus à la famille des contrats de composition externe, mais peuvent être vus comme des contrat de composition interne à l'intérieur du composite *ROOT*.

– *Spécification 2* : On imagine également que pour des raisons de développement, on doit impérativement utiliser une version de java > 1.5 - le programmeur a utilisé les versions Template des classes du package *java.util* -. Cette contrainte de déploiement est posée sur le *ROOT* et permet de s'assurer que l'assemblage fonctionnera correctement dans l'environnement d'exécution :

```
on ROOT
    inv : <*>.attributes.getJdkVersion() >= 1.5
        // attribut JdkVersion
        // sur le composant ROOT
```

– *Spécification 3* : On suppose que les deux lampes sont liées à l'interrupteur. On souhaite vérifier que lorsque l'on appuie sur le bouton *on*, les lampes ont changé d'état. On veut également s'assurer que le petit compteur a bien été incrémenté, mais reste inférieur à la valeur M. En effet, les lampes utilisées deviennent dangereuses après M éclairages. Il est alors préférable de ne pas les utiliser.

```
on <ROOT>
    context void i1.on()
      pre : l1.timeOfUse() < M && l2.timeOfUse() < M
                //vérifie le nombre d'utilisations
      post : l1.isSwitechedOn() && l2.isSwitchedOn()
                //les lampes ont été allumées
             && l1.timeOfUse() = l1.timeOfUse()@pre + 1
             && l2.timeOfUse() = l2.timeOfUse()@pre + 1
                //le compteur a été incrémenté
  context void i1.off()
    post : !l1.isSwitechedOn() && !l2.isSwitchedOn()
```

Mise en oeuvre de la contractualisation

Le système de contractualisation crée des contraintes à partir des spécifications. Il est composé :
- du **référentiel de spécifications** : il conserve les spécifications liées à chaque composant et qui sont nécessaires pour construire le contrat ;
- du **contrôleur de contrats** *ContractController* : c'est l'identité responsable de gérer les contrats. Pour cela, il communique avec d'autres contrôleurs de base de Fractal, notamment le *ContentController* et le *BindingController*. Il s'appuie sur le service d'exécution de contrats pour réagir aux actions de connexion ou de composition et ainsi déclencher les activités du système de contrats. Il est propre à chaque composant et possède le même statut que les contrôleurs de base de l'implémentation Fractal ;
- du **service d'interception d'exécution de contrats** : il propose une API d'interception pour informer le contrôleur de contrat des activités du système correspondant.

Dans ConFract, les contrats sont construits pendant la phase d'assemblage des composants en utilisant les descriptions de schémas d'architecture, de signature d'interface et de spécifications. Les spécifications sont ainsi clairement séparées des contrats.

Au cours de réification d'un contrat, le métamodèle de ConFract détermine les responsabilités associées à chaque spécification, parmi la liste des participants du contrat. Lorsque les responsabilités d'une spécification sont toutes déterminées, celle-ci devient une **disposition** du contrat. Les responsabilités peuvent être :
- **garant** : c'est l'unique composant qui doit être prévenu en cas de négation de la disposition et qui a la capacité d'agir pour traiter le problème
- **bénéficiaire** : les composants bénéficiaires sont ceux qui peuvent compter sur la véracité de la disposition et qui peuvent être prévenus en cas de changement d'état de celle-ci (négation ou rétablissement)
- **contributeur** : il s'agit des composants qui participent à la véracité d'une disposition, et à qui on peut demander de « faire un effort » pour le rétablissement de la disposition.

Deux phases caractérisent le fonctionnement du système de contractualisation :
- la **phase de configuration** qui réalise la construction des contrats : le contrôleur de contrat est prévenu, par le contrôleur de connexion ou de composition, de l'admission d'un nouveau composant. Il initialise la création d'un contrat et s'appuie sur le référentiel de spécification pour obtenir les spécifications de chaque composant. À ce stade, le contrat est dit ouvert : les participants du contrat restent inconnus et les spécifications peuvent changer. Le contrat devient fermé lorsque tous les participants sont connus et que toutes les spécifications sont stabilisées. Dans ce cas, ni spécification, ni garant ne peuvent être modifiés, en revanche le contrat peut être annulé et recréé. La fermeture du contrat correspond, dans la vie quotidienne, à la signature effective du contrat.
- la **phase de vérification des contrats** : pendant cette phase, chaque spécification du contrat fermé est vérifiée une à une. Si une spécification n'est pas satisfaite alors le contrat est violé. En fonction des

propriétés qu'elle exprime, une spécification peut être vérifiée à la fin de la configuration des composants ou à la fin de l'exécution. Ainsi lors de la vérification, on peut déterminer la compatibilité ou l'incompatibilité à la configuration, ou alors ne pas pouvoir décider de la compatibilité statiquement et devoir attendre l'exécution pour pouvoir se prononcer.

La figure Figure 2.7 donne une vision schématique du processus de contractualisation.

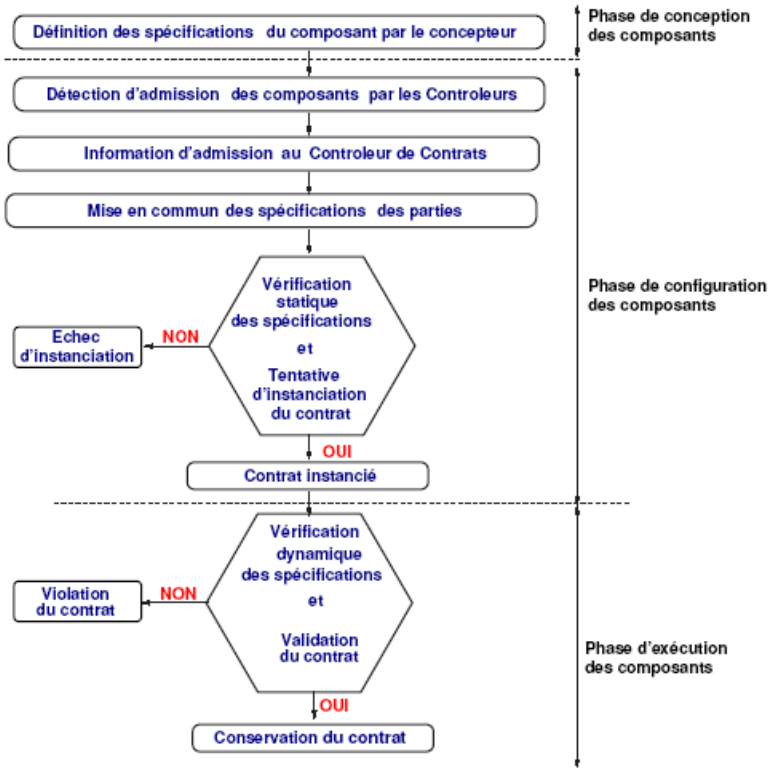

Fig. 2.7 – Schéma général du processus de contractualisation dans ConFract

2.4.3 Conclusion sur Fractal/ConFract

Fractal est une plateforme intéressante de par son modèle à base de composites et de composants de base qui permet de parfaitement hiérarchiser les parties d'une application.

Cependant, la partie qui nous importe est son extension ConFract. Poser des contrats pour obtenir un comportement particulier est parfaitement en accord avec les objectifs que l'on s'est fixé.

Il est cependant clair que le système de gestion de contrat est bien trop compliqué pour pouvoir être appliqué à la plateforme Wcomp, qui est la plateforme qui nous intéresse. De plus, il est pertinent de préciser ici que lorsqu'un contrat est violé, le principe de résolution de conflit passe par une négociation entre les différents partis, processus qui peut être long et qui peut ne pas aboutir. Ceci ne convient pas à une application temps réel comme celle que nous souhaitons envisager. Nous reviendrons sur ce point dans la partie 3.

2.5 UPnP – *Universal Plug and Play*

Les paragraphes[8] qui suivent sont une courte introduction à une technologie émergente promue en partie par Intel et Microsoft. J'ai eu recours à cette technique pour la mise en place de contrats réagissant au runtime (cf partie 3.4).

UPnP est un ensemble de protocole réseau promulgué par un forum de normalisation dédié[9]. Le but d'UPnP est de faire des équipements dotés de comportements génériques afin de simplifier l'installation et l'interopérabilité des équipements susceptibles de communiquer avec d'autres équipements. UPnP réalise ce challenge en établissant des protocoles de commandes d'appareils s'appuyant sur des standards de communication utilisés dans l'univers d'Internet : TCP/IP ainsi que HTML/XML.

2.5.1 Principe

UPnP repose sur le principe des clients/fournisseurs de services : chaque élément implémentant UPnP sera soit un point de contrôle (un client), soit un serveur (un fournisseur) ou les deux.

Lors de la mise en réseau d'un fournisseur de services, celui-ci va annoncer sa présence sur le réseau avec le type de services qu'il fournit et un moyen de le contacter. Les points de contrôle déjà présents sur le réseau sont alors informés de l'ajout d'un équipement réseau fournissant un service et pourront alors mémoriser ces informations s'il s'avère que l'équipement est en mesure de faire appel aux services du fournisseur.

2.5.2 Fonctionnement d'UPnP

UPnP est protocole réseau. Son fonctionnement est décomposé en six phases intervenant à différents niveaux du modèle OSI[10]. Les principes mis en œuvre dans UPnP sont complets et regroupent les mécanismes d'auto-configuration, au niveau des couches basses du réseau, ainsi que les protocoles de communications inter-applications.

- **Adressage** : Lors de la mise en service d'un périphérique, celui-ci va chercher à disposer d'une adresse IP afin de pouvoir dialoguer avec les éventuels autres périphériques déjà présents. Pour cela il utilise le protocole DHCP (Dynamic Host Configuration Protocol) afin de demander l'assignation d'une adresse IP qui lui sera réservée auprès d'un serveur DHCP. Si aucun serveur DHCP n'est présent ou qu'il ne reçoit pas de réponse, le périphérique UPnP devra faire appel à une méthode d'auto-désignation d'adresse IP appelée Auto-IP. Auto-IP reprend le fonctionnement apparu dans Microsoft Windows depuis Windows 98 en mode de configuration automatique (client DHCP) en l'absence de serveur ou réponse DHCP : il s'agit ni plus ni moins d'APIPA (Automatique Private IP Addressing).

- **Découverte** : La phase de découverte permet aux périphériques UPnP de localiser les périphériques intéressants (qui sont susceptible d'entrer en interaction avec le périphérique en question), d'annoncer sa présence, ainsi que les services qu'il offre aux autres périphériques du réseau. Dans les deux cas, il s'agira de petits messages de découverte contenant des informations basiques sur les services offerts comme leur type et leur identifiant.

[8] Les sources ayant permis l'écriture de la partie UPnP sont :
- Comment ça marche : l'Universal Plug and Play : `http://www.01net.com/editorial/272988/je-branche-et-...-c-est-tout/comment-ca-marche-l-universal-plug-and-play/`
- Universal Plug and Play : `http://en.wikipedia.org/wiki/Universal_Plug_and_Play`
- Universal Plug-and-Play : `http://supinfo-projects.com/fr/2004/upnp/1/`

[9] `http://www.upnp.org/`

[10] Le modèle d'interconnexion des systèmes ouverts de l'ISO (International Standardisation Organization – Organisation internationale de normalisation) est un modèle de communications entre ordinateurs. Il décrit les fonctionnalités nécessaires à la communication et l'organisation de ces fonctions. *Source : Wikipédia – `http://fr.wikipedia.org/w/index.php?title=Mod\%C3\%A8le_OSI&oldid=9857649`*

Lors de sa connexion, un périphérique va propager son identité sur le réseau en mode multi-cast, et éventuellement émettre un message de découverte. À la réception de celui-ci, les périphériques répondront à l'émetteur en lui communiquant leur signature.
Un point de contrôle peut donc apprendre la présence de périphériques intéressant par le biais de deux moyens : à l'aide d'une recherche de sa part ou grâce à l'annonce automatique d'un périphérique.
De manière similaire à l'annonce lors de la connexion à un réseau, un périphérique UPnP doit annoncer sa cessation de service sur le réseau lorsqu'il est sur le point d'être retiré de celui-ci.
Tant qu'un service ou périphérique n'a pas annoncé sa fin de disponibilité, tout point de contrôle suppose qu'il est toujours présent et disponible.

- **Description** : Une fois qu'un point de contrôle ou client a localisé un service intéressant grâce à la phase de découverte, il peut procéder à la phase de description qui lui permettra d'interroger le service afin de connaître ses caractéristiques. En effet, la phase de découverte n'apprend que peu d'informations aux divers points de contrôle à propos d'un service : le type de service ou périphérique UPnP, un identifiant, une URL et une description. Pour en savoir plus sur le périphérique ou service, et pour interagir avec lui, le point de contrôle doit d'abord récupérer les informations du périphérique comme ses capacités à l'aide de l'URL récupérée durant la phase de découverte.
  La description UPnP d'un périphérique est divisée en deux parties :
  – la description du périphérique et de ses conteneurs logiques
  – la/les descriptions des services proposés avec leurs capacités respectives.
  Les messages de description sont au format XML et définis par les constructeurs selon un modèle défini par les spécifications UPnP définies par l' UPnP Forum.
  La description d'un service UPnP est une liste de commandes ou action que le service prend en charge ainsi que les paramètres ou arguments nécessaires. La description d'un service comprends également une liste de variables décrivant l'état du périphérique. Chaque variable est décrite par un type de données, un intervalle de validité et caractéristiques d'événement.
  La récupération du message de description par le point de contrôle est déclenchée par le point de contrôle, à l'aide une requête HTTP de type GET à l'URL annoncée dans le message de découverte. Il en est de même pour les différents services que comprend un périphérique.

- **Contrôle** : Un point de contrôle va être en mesure de demander des actions de la part d'un service. Il s'agit du principe Remote Procedure Call (RPC). Le point de contrôle envoie une commande et éventuellement des paramètres à un service qui tentera d'effectuer la tâche. Une fois la tâche accomplie (ou échouée), le service renvoie les résultats ou erreurs.
  Il en est de même pour les variables d'état des périphériques et service, un point de contrôle peut à tout moment connaître leur valeur à l'aide d'une requête de contrôle.
  L'architecture UPnP définit la manière de communication utilisée pour faire transiter ces échanges. Les messages sont délivrés via requêtes HTTP à l'aide de TCP/IP et sont formatés en format XML SOAP.

- **Événements** : La phase de gestion des événements permet aux périphériques de s'adapter aux changements de topologie du réseau ainsi qu'aux changements d'états des divers périphériques.
  Un point de contrôle peut alors s'abonner auprès d'un service, et plus particulièrement à l'une de ses variables afin de pouvoir être tenu au courant des éventuelles modification de sa valeur.
  Lors d'une demande d'abonnement, le service peut, suivant des critères qui lui sont propres, accorder l'autorisation ou refuser l'abonnement. Si l'abonnement est accepté, le message d'acceptation est accompagné d'un délai durant lequel l'abonnement restera actif, ainsi que de la valeur initiale de la variable.
  Il est de la responsabilité du point de contrôle de renouveler l'abonnement après le délai imparti. Si le demandeur n'a plus besoin de suivre l'évolution d'une variable faisant l'objet d'un abonnement, il doit demander au service la résiliation de son abonnement.

- **Présentation** : Une fois qu'un point de contrôle a découvert et récupéré les informations d'un autre périphériques UPnP, il peut mettre en place la présentation de celui-ci et afficher une interface de visua-

lisation/administration. Cette interface se base ici sur un transfert HTTP. Le langage de présentation est le HTML.
Le standard UPnP ne donne pas d'instructions détaillées sur les spécifications de la phase de présentation : les constructeurs sont libres pour la conception de cette interface.

2.5.3 Apports d'UPnP

L'intérêt principal d'UPnP est son fonctionnement proche de celui des services web puisque chaque serveur UPnP publie la liste de ses fonctionnalités et que chaque point de contrôle peut y acceder indépendamment de son langage d'implémentation.

Le principal avantage de cette technologie réside donc dans sa capacité d'émettre des événements auxquels peuvent s'abonner les points de contrôles. On fait très rapidement le lien avec Wcomp où un composant possède des fonctions, qui sont ses ports d'entrées, et qui génère des événements, qui sont ses ports de sortie.

L'intérêt secondaire est le protocole d'entrée/sortie dans le réseau puisque chaque serveur UPnP entrant se signale auprès des autres composant UPnP. Cette propriété est exploité par la plateforme Wcomp qui génère un composant Wcomp pour chaque serveur UPnP entré sur le réseau et l'ajoute à la liste des composants connus.

2.6 Conclusion

Avant de continuer et proposer ma solution aux problématiques introduites dans la partie 1.1, faisons le point sur les apports de notre état de l'art.

Les modèles à composants permettent de modéliser un mode de programmation basé sur le regroupement par fonctionnalités. Ils apportent une grande souplesse en terme de configuration et reconfiguration des applications "à chaud", permettant une adaptation à l'environnement sans avoir à relancer l'application.

En parallèle, le modèle d'interactions permet de faire communiquer des composants dans un environnement hétérogène. Il autorise la spécification des interactions entre les composants indépendamment de l'application finale, et établit, après fusion des schémas d'interactions, d un assemblage en tenant compte des définitions de l'utilisateur.

À la suite de nos recherche, nous pouvons conclure qu'il semble intéressant d'allier les avantages de Wcomp et de ConFract. Wcomp, plateforme basée sur un modèle à composants, peut utiliser un modèle d'interactions, et est capable de gérer des composants mixtes, c'est à dire des composants hardware embarquant un module leur permettant de répondre à des requêtes logicielles. ConFract, quant à elle, permet la gestion de contrats de compositions et permet de poser des contraintes sur les assemblages.

L'application que nous voulons mettre en place est basée sur un assemblage se configure et reconfigure automatiquement et le plus rapidement possible en réponse au changement dans l'environnement (entrée/-sortie d'un nouveau composant). Dans cette optique il est donc important que les processus d'assemblage et de résolution de contrats, aient toujours le moyen d'aboutir. Les contrats envisagés ne sont donc pas du même ordre que sur la plateforme ConFract (partie 2.4.2) puisqu'il est nécessaire d'offrir la solution souhaitée au conflit que pourrait générer les contrats.

Notre but dans la suite va donc être d'allier les avantages de ConFract et de Wcomp afin d'élaborer la solution à nos scénarios d'utilisation. Pour cela, on aura recours à la technologie UPnP qui est un protocole réseau, très utilisé avec la plateforme Wcomp, et qui permet de réagir dans un environnement doublement hétérogène, avec des composants logiciels et matériels utilisant des langages différents (Java et C#).

Chapitre 3

Ma proposition

Au cours de cette partie, après avoir isolé quelque notions préliminaires, je vais décrire les premières idées qui me sont venues à la suite de mon tour d'horizon sur les travaux déjà effectués, en particulier sur ISL et CCL-J. Je détaillerai mes premiers pas dans le cœur du sujet, et les erreurs qui m'ont permis d'aboutir à la création de deux nouveaux types de contrats, les contrats d'ensemble et les contrats avec gardien, que je détaillerai dans des parties disctinctes. J'exprimerai dans la dernière partie les différences notables entre mes contrats et les contrats rencontrés sur la plateforme ConFract.

3.1 Notions préliminaires à mon étude

Reprenons une fois encore l'exemple décrit dans la section 1.1. Au vu de l'état de l'art, nous pouvons poser notre problématique dans ces termes :

Soit un **ensemble d'interactions potentielles**, c'est à dire que des descriptions d'interactions entre composants ont été faites par l'utilisateur et que notre application doit tenir compte de ces définitions.

Pour former une application, un sous-ensemble d'interactions potentielles doit-être **sélectionné** et **composé**. Formulé différemment, seule une partie des interactions spécifiées par l'utilisateur vont-être prises en compte par le système pour pouvoir configurer le programme dynamiquement en fonction de l'environnement d'exécution.

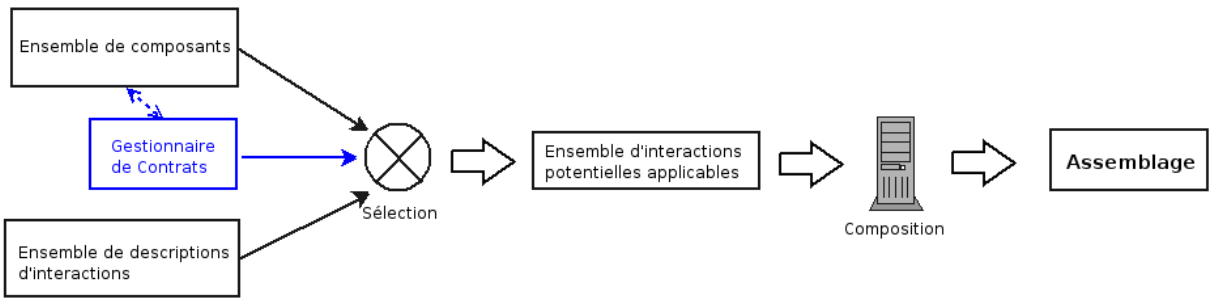

FIG. 3.1 – le modèle global de notre étude

À partir de ces postulats, nous souhaitons mettre en place des contrats qui vont modifier l'assemblage en fonction des composants présents, ainsi que d'événements pouvant survenir durant l'exécution. Un **gestionnaire de contrats** (cf figure 3.1) va donc avoir pour rôle de gérer cet ensemble de contraintes en modifiant le sous-ensemble des interactions potentielles applicables.

Ce sont ces conclusions que je vais illustrer et justifier dans la suite.

3.2 Parallèle entre CCL-J et ISL

À la suite de l'étude successive d'ISL et de ConFract, un certain nombre d'analogie entre CCL-J et ISL me sont apparues :

- ils régissent les comportements des composants les uns par rapport aux autres
- ils expriment l'envie de séparer des informations transverses du corps de l'application, les contraintes sur les interactions dans le cas de CCL-J, la description même de ses interactions pour ISL
- ils utilisent tout deux des langages très expressifs, autorisant des structures de contrôle et en particulier des structures conditionnelles.

L'une des premières idées a donc été de tenter de les composer. Ainsi, à partir d'un contrat CCL-J à base de pré et de postcondition, et d'un contrat ISL ; je voulais obtenir un schéma ISL exprimant les contraintes du contrat en CCL-J.

Le contrat CCL-J

```
context  void l1.on()
    //precondition : la lampe n'est pas déjà éclairée.
    pre : !l1.isSwitchedOn()
    //postcondition : la lampe est éclairée.
    post: l1.isSwitchedOn()
```

Le schéma ISL

```
Pattern p1 (Lampe l1, Interrupteur i1){
    i1.^on()  :- i1._call  // l1.on()
}
```

Le contrat mixte résultant de la composition ISL – CCL-J

```
Pattern p1 (Lampe l1, Interrupteur i1){
    i1.^on()  :-
        if(!l1.isSwitchedOn()){        //precondition
            i1._call  || l1.on()        //appel
            if(!l1.isSwitchedOn()){   //postcondition
                ERREUR
            }
        } else {
            ERREUR
        }
}
```

L'idée première de cette démarche était de montrer qu'ISL était probablement assez riche pour exprimer les contraintes du contrat CCL-J. Ces résultats semblaient prometteurs, mais de nombreux problèmes montraient que l'on était pas tout à fait sur la bonne voie :

- on ne savait pas vraiment que faire en cas d'erreur.
- dans le cas de composition plus complexes, avec par exemple deux schémas d'interaction composables, il se posait alors la question de la commutativité entre les compositions entre schémas ISL et avec le contrat CCL-J, ainsi que sur l'équivalence de la sémantique résultante avec celle exprimée par les contrats et schémas "purs"
- les contrats CCL-J peuvent travailler autant sur les instances nommées de composant que sur les types, tandis que ISL ne travaille qu'au niveau des types.
- CCL-J permet la définition d'invariants (soit au niveau de la méthode, soit au niveau de toute l'application). Avec ce genre de composition il était impossible d'exprimer de telles conditions.

Une fois montré qu'ISL n'était pas suffisant expressif, nous avons essayé de nous servir de la combinaison des deux pour exprimer des schémas plus simple.

Le schéma ISL "complexe". La fonction changeState() permet d'éteindre les lampes si elles sont allumées et de les allumer dans le cas contraire.

```
Pattern p1 (Lampe l1, Interrupteur i1){
     i1.^changeState()  :-
          if(l1.isSwitchedOn()){
                i1._call  || l1.off()
          } else {
                i1._call  || l1.on()
         }
}
```

Les schémas ISL simple résultat et le contrat CCl-J les liants

ISL

```
Pattern p2 (Lampe l1, Interrupteur i1){
     i1.^changeState()  :-  i1._call  || l1.off()
}

Pattern p3 (Lampe l1, Interrupteur i1){
     i1.^changeState()  :-  i1._call  || l1.on()
}
```

CCLJ

```
context  void i1.on()
     pre:  if(l1.isSwitchedOn()){
                p2.activate() || p3.unactivate()
           } else {
                p2.unactivate() || p3.activate()
           }
```

L'idée de la démarche, à l'inverse de la première, était de montrer que l'on pouvait se retreindre à des schémas ISL sans condition et que CCL-J permettait de remplacer la complexité d'ISL. Mais là encore, on pouvait se poser des questions au niveau de la sémantique résultante, mais aussi sur l'intérêt de la démarche puisque le nombre de ligne à écrire, pour une sémantique que l'on veut équivalente, est plus important (donc plus de risque d'erreur).

Au vu de ces résultats, je me suis penché sur le modèle global de notre problème et y ai introduit la notion de contrat tel que je la concevais (cf. figure 3.2). Un contrat réagit à un contexte donné en fonction de son environnement. Il modifie alors l'ensemble des Interactions Potentielles qui peuvent apparaître dans le système. L'ensemble des Interactions Potentielles réellement instanciées est modifié selon le contexte.

À partir de ce modèle, et pour résoudre les scénarios d'utilisation décrits dans la partie 1.1, j'ai introduit deux catégories de contrats que je vais détailler dans les sections suivantes : les contrats d'ensemble et les contrats avec gardien.

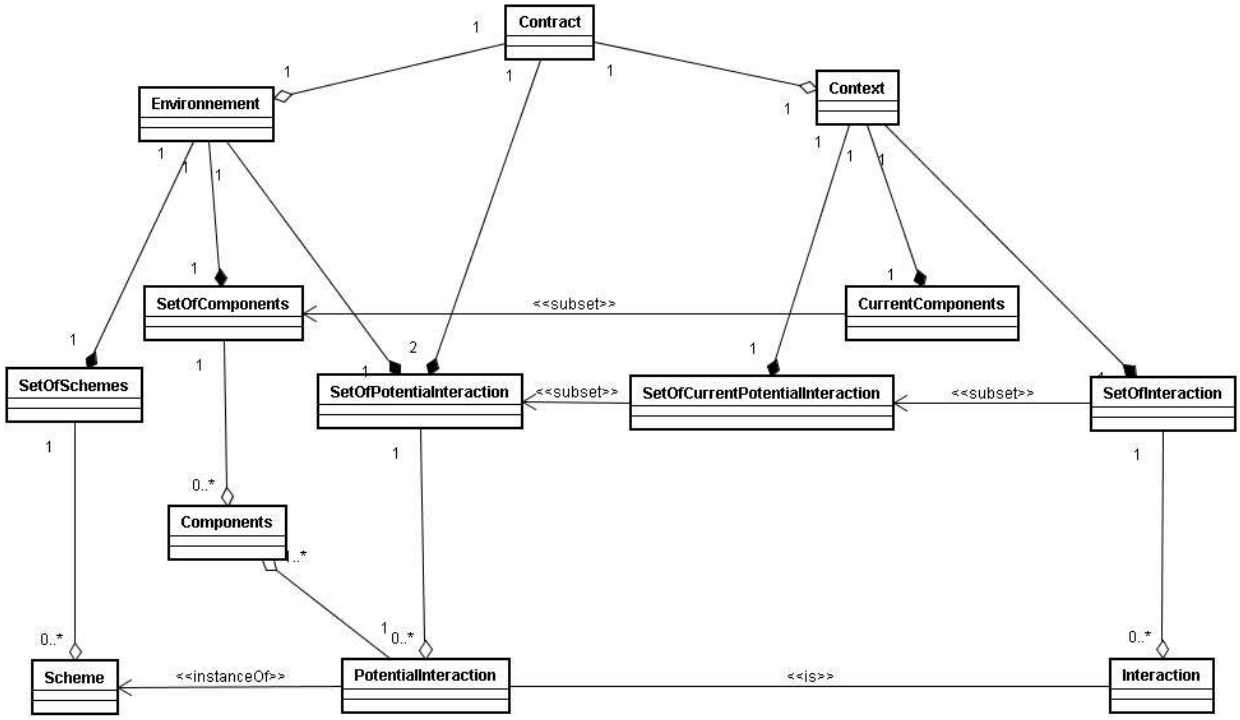

Fig. 3.2 – le modèle wcomp étendu aux contrats

3.3 Les contrats sur des ensembles d'interactions potentielles (contrat d'ensemble)

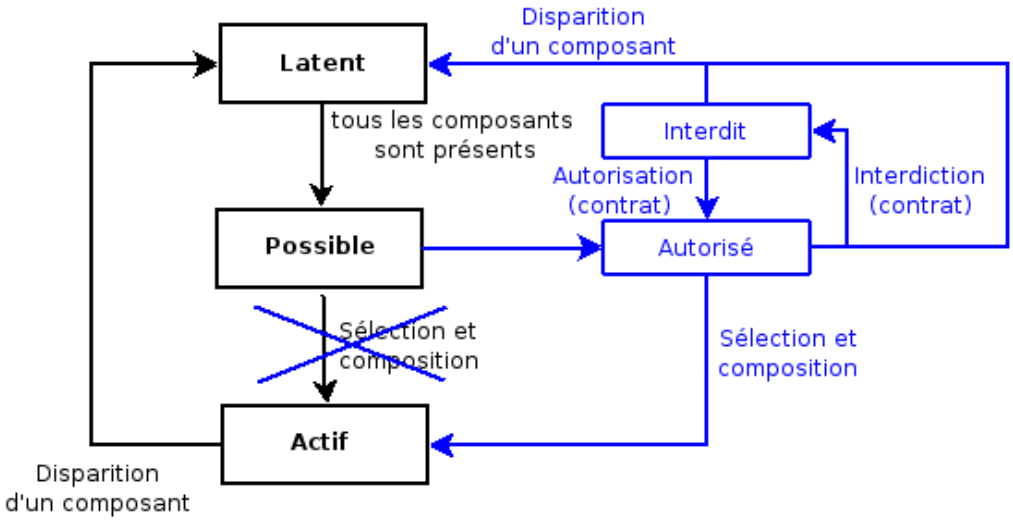

Fig. 3.3 – Graphe de changement d'états d'une interaction potentielle

Commençons par présenter les contrats d'ensemble. Ces contrats ont pour but de modifier l'ensemble des interactions potentielles. Il est notable que ces dernières passent par plusieurs états successifs :

- **Latent** : A cette étape, l'utilisateur a spécifié l'existence d'une interaction avec le formalisme ISL4Wcomp décrit dans la partie 2.3.1.
- **Possible** : L'ensemble des composants intervenants de l'interaction sont présents dans le système.
- **Actif/Instancié** : L'interaction a été retenue par le système, et est effectivement présente dans l'assemblage.

Nous allons rajouter deux états **Interdit** et **Autorisé** qui vont symboliser l'action des contrats, et le blocage conditionnel du passage entre l'état Possible et l'état Actif. La figure 3.3 illustre l'ajout de ces deux états(en bleu). Le passage par composition et selection de l'état Possible à Actif (en noir) est remplacé par un passage automatique de l'état Possible à Autorisé. De là, l'interaction potentielle peut passer dans l'état interdit en fonction du gestionnaire de contrats (qui peut aussi autoriser une interaction préalablement interdite) ou dans l'état Actif si l'interaction est sélectionnée et composée.

Ce travail fait par le gestionnaire de contrat est donc de passer chaque interaction dans l'état Autorisé ou Interdit (si elle est sortie de l'état Latent). Une fois ce travail fait, le sélecteur n'a plus qu'à collecter les interactions dans l'état Autorisé et les envoyer au composeur qui va générer l'assemblage.

3.3.1 Les contrats de dépendance

Je vais maintenant revenir sur les premiers scénarios d'utilisation, introduits dans la partie 1.1, qui ne pouvaient pas être pris en compte en l'état initial du projet.

Le premier scénario d'utilisation est très simple : on souhaite s'assurer que les lampes l_1 et l_2 seront bien synchronisées.

La contrainte que l'on pose ici est que si l'interaction potentielle *int_lam1* est applicable, alors on souhaite que l'interaction potentielle *int_lam2* le soit aussi (idem pour *pad_lam1* et *pad_lam2*). Si l'on se réfère au Graphe de la figure 3.3, ce que nous voulons exprimer est que les interactions potentielles *int_lam1* et *int_lam2* doivent être dans le même état (Autorisé ou Interdit).

On peut donc envisager l'écriture d'un contrat garantissant cette dépendance. Écrivons les contrats entre l'interrupteur et les lampes, ainsi qu'entre le PDA et ces mêmes lampes de la façon suivante :

Contrat entre entre le PDA et les lampes

```
contract type Dependance{
     depends pda_lam1, pda_lamp2 ; //la liste des interactions
                                   //potentielles devant être réunies
                                   //en même temps dans le système
}
```

Contrat entre entre l'interrupteur et les lampes

```
contract type Dependance{
      depends int_lam1, int_lamp2 ;
}
```

3.3.2 Les contrats d'exclusion mutuelle

Le second scénario d'utilisation que nous pouvons introduire est le suivant :

Nous souhaitons que lorsque le PDA est présent dans le système, ce soit lui qui contrôle les lampes et non pas l'interrupteur mural.

Celà revient à dire que l'on souhaite que lorsque les interactions potentielles *pda_lam1* et *pda_lam2* sont applicables, alors les interactions potentielles *int_lam1* et *int_lam2* ne doivent pas l'être. Encore une fois, relativement à la figure 3.3, cela signifie que les interaction *pda_lam1* et *int_lam1* ne peuvent pas être simultanément dans le même état Autorisé ou Interdit (idem pour *pda_lam2* et *int_lam2*).

Cette affirmation revient à introduire une notion d'exclusion mutuelle entre *int_lamp1* et *pda_lamp1*. Cependant, comme nous sommes dans un milieu dynamique, il est nécessaire de prévoir le comportement à adopter si les deux interactions potentielles en exclusion sont simultanément applicables. Il semble donc important d'introduire une notion de priorité entre ces interactions potentielles, qui designera, a priori, celle qui sera alors sélectionnée en cas de conflit.

Nous pouvons donc écrire des contrats d'exclusion mutuelle sous la forme :

Contrat d'exclusion entre l'interrupteur et le PDA pour la lampe l_1

```
contract type ExclusionMutuelle {
      exclude int_lam1, pda_lam1; //les Interactions potentielles
                                  //en exclusion mutuelle
      priority pda_lam1; //celle qui est prioritaire
}
```

Contrat d'exclusion entre l'interrupteur et le PDA pour la lampe l_2

```
contract type ExclusionMutuelle {
      exclude int_lam2, pda_lam2;
      priority pda_lam2;
}
```

3.3.3 Composition des différents types de contrat sur les ensembles

Une fois introduite les notions d'exclusion mutuelle et de dépendance, il n'est pas inutile d'étudier la notion de dépendance entre eux. En effet, il semble relativement infondé de vouloir les étudier indépendamment, puisqu'il est fortement probable qu'une application un peu complexe ait besoin d'utiliser les deux notions.

Si l'on regarde notre exemple de plus près, on remarque d'ailleurs qu'il est déjà intéressant de les considérer conjointement. Dans notre second scénario d'utilisation, qui nous a conduit à introduire la notion d'exclusion mutuelle, il est parfaitement compréhensible de vouloir utiliser la notion de dépendance simultanément. La logique veut que les actions d'éclairer et d'éteindre les lampes se fassent du même point d'accès.

Dans les prochains paragraphes, je vais donc étudier l'utilisation conjointe de ces types de contrats et les mécanismes de résolution de ses contrats afin d'obtenir un ensemble d'Interactions Potentielles validant l'ensemble des contrats posés.

Je vais cependant faire l'hypothèse qu'il est plus pertinent de traiter indépendamment chaque type de contrat.

Ordre de composition

Dans l'optique de résolution par type de contrat, il est important d'étudier si l'ordre de résolution est important.

Prenons un ensemble d'interactions potentielles $\{A_1, A_2, A_3, A_4, A_5, A_6\}$. Supposons deux contrats d'exclusion mutuelle $A_1 > A_3$[1] et $A_4 > A_6$; ainsi que deux contrats de dépendance $A_1 - A_2$[2] et $A_3 - A_4$.

[1] le ">" symbolise l'ordre de priorité entre les 2 partis en exclusion mutuelle. Le parti marqué comme supérieur sera sélectionné en cas de conflit au détriment de l'autre.

[2] le "-" symbolise la dépendance entre les deux interactions potentielles. Ce contrat est validé si les deux partis sont presents dans l'ensemble sélectionné. Si seulement un des deux parti est présent dans cet ensemble, alors il sera retiré de l'ensemble des sélectionnes

Appliquons[3] tout d'abord l'exclusion mutuelle, puis la dépendance :

$$\dfrac{\dfrac{\{A_1, A_2, A_3, A_4, A_5, A_6\} \qquad A_1 > A_3 \qquad A_4 > A_6}{\{A_1, A_2, A_4, A_5\}} \qquad A_1\text{-}A_2 \qquad A_3\text{-}A_4}{\{A_1, A_2, A_5\}\ (Res_1)}$$

Appliquons maintenant ces deux opérations dans l'ordre inverse :

$$\dfrac{\dfrac{\{A_1, A_2, A_3, A_4, A_5, A_6\} \qquad A_1 - A_2 \qquad A_3 - A_4}{\{A_1, A_2, A_3, A_4, A_5, A_6\}} \qquad A_1 > A_3 \qquad A_4 > A_6}{\{A_1, A_2, A_4, A_5\}\ (Res_2)}$$

L'étude de cet exemple simple nous montre que les opérations ne sont absolument pas commutatives. L'ordre dans lequel vont-être fait les opérations a donc une importance. Cependant, je ne me pencherais pas en détail sur ce problème et dans l'implémentation de cette composition (cf partie 4.1.2, je choisirai le second ordre de composition.

Cependant, on constate sur cet exemple que $Res_1 \subset Res_2$. Ce résultat est à rapprocher des travaux de Tom Mens et Günter Kniesel ([15]) qui concluent que tous les algorithmes de transformations ou réécritures de graphes définissent automatiquement des priorités. Le principe est basé sur la differentiations entre les opérations ayant des effets positifs, et celles ayant des effets négatifs. L'application de la dépendance permet la création d'ensembles les plus grands possibles (effet positif), tandis que les exclusions mutuelles diminuent l'ensemble des interactions autorisées (effet négatif). Le plus grand ensemble est obtenu lorsque l'on applique tous les effets positifs avant les effets négatifs. L'ensemble Res_2 est donc le plus grand ensemble d'interactions autorisées possible. Il semble naturel de privilégier le maximum d'interactions entre composants afin de maximiser les fonctionnalités, cet ordre de composition est donc logique.

3.4 Les contrats de réaction au runtime (Contrat avec gardien)

Dans cette partie, je vais introduire une nouvelle famille de contrats, en solution de notre dernier scénario.

Dans la section 1.1 nous avons introduit un détecteur de présence qui n'a pas encore trouvé d'utilité. Son utilisation apparaît dans la partie 1.1, avec le troisième scénario d'utilisation : comment faire pour n'allumer qu'une lampe en fonction de la présence ou non d'une personne dans le champ du détecteur (i.e. Allumer l_1 si le détecteur indique la présence d'une personne, l_2 sinon).

La solution que je propose est l'introduction d'un nouveau type de contrat que j'appellerai **contrat avec gardien**. Ces contrats sont d'une forme inspirée du langage CCL-J introduit dans la section 2.4.2.

On s'aperçoit, à la vue de cet exemple, que ce type de contrat est différent de ceux vus dans la section 3.3. En effet, ils font intervenir le notion de déclencheurs pour l'évaluation. Si les contrats sur les ensembles étaient évalués au moment de modifications sur l'ensemble de composants présents ; ces contrats spécifient le moment de leur évaluation, c'est à dire le signal que le système doit capter pour lancer le processus de traitement du contrat. Ici, le système doit déclencher l'évaluation de le condition au moment où le pda ou l'interrupteur envoie le signal *on*.

[3] Le trait horizontal symbolise l'opération de résolution de l'ensemble pour qu'il obéisse aux contrats donnés

Exemple[4] *de contrat avec gardien, permettant de mettre en oeuvre le scénario d'écrit précédemment.*

```
contract type WithGuardian { //type de contrat
    context {i1.^on() | pda1.^on()} //moment d'évaluation
    if (d1.HasSomeone?()){
        // actions à effectuer si la condition est vrai
        select int1_lam1, pda1_lam1; //IP à sélectionner
        deselect int1_lam2, pda1_lam2; //IP à déselectionner
        error « Seule la lampe 1 est commandée »
                    //message d'erreur à afficher
    } else {
        // actions à effectuer si la condition est fausse
        select int1_lam2, pda1_lam2;
        deselect int1_lam1, pda1_lam2;
        error « Seule la lampe 2 est commandée »
    }
}
```

On peut remarquer ici une similitude avec les contrats mixtes résultant de le composition d'ISL et CCL-J que nous avions introduit dans la partie 3.2. Nous pouvons en effet supposer en quelque sorte que nous avons composé les deux fichiers suivants.

ISL

```
Pattern p1(Interrupteur i1, Lampe l1,
           Lampe l2, PDA pda1, Detecteur d1){
    i1.on() -> i1._call
}
```

CCLJ

```
context void i1.on()
    pre : d1.HasSomeone?()
```

La richesse de ce contrat est de fournir en plus le comportement à adopter en cas de validation ou d'infirmation de la précondition.

3.5 Différences avec les contrats ConFract

Pour illustrer nos propos, tentons de dresser un tableau comparatif (tableau 3.1) entre les différents types de contrats que nous avons rencontrés.

Au vu de ce comparatif, on s'aperçoit en fait que les contrats introduits dans ConFract et ceux qui correspondent à ma solution n'ont que peu de points communs.

La principale différence, qui induit en partie toutes les autres, est que les contrats d'ISL4Wcomp modifient l'assemblage entre les composants. En fractal, ce n'est pas le cas : en cas de violation de contrat c'est le processus de négociation entre les différents participants qui permet une résolution de conflit. Cette différence notable engendre une différence profonde entre les deux plateformes.

[4] dans les commentaires, IP : Interaction Potentielle

	ConFract		**ISL4Wcomp évolué**	
	Contrat d'interface	**Contrat de composition**	**Contrat d'ensemble**	**Contrat avec gardien**
Notion de contexte (méthode ou interface)	X	X		X
Gestion de contrat par type de composants	X	X		
Gestion de pré et postcondition	X	X		
Contrat avec spécification de correction			X	X
Contrat avec correction par négociation	X	X		
Évaluation des contrat durant le runtime	X	X		X
Modifie l'assemblage			X	X

Tab. 3.1 – Quelques différence entre type de contrats

Chapitre 4

Implémentation de la solution

Au cours de cette dernière partie, je vais détailler quelques points phares de l'implémentation des solutions exposées dans la partie précédente. Afin de pouvoir éventuellement généraliser mon implémentation, je vais isoler les processus et algorithmes ayant trait au modèle ; puis j'expliciterai certaine contraintes liées à Wcomp avant d'offrir une vue d'ensemble de ma solution.

4.1 Détails d'implémentation au niveau du modèle

La pièce maîtresse de notre solution est le gestionnaire de contrat. C'est lui qui contient la liste de toutes les contraintes posées par l'utilisateur, et c'est donc lui qui a la charge de faire varier l'ensemble des interactions potentielles applicables et autorisées. Il est donc bon de s'intéresser à quel moment il est opportun de traiter les différents types de contrats.

Pour mener à bien sa tâche, le gestionnaire de contrat doit aussi être en mesure de résoudre les conflits générés par le nombre des contrats. D'autant que ces contrats ne sont pas tous du même type. Je vais donc également détailler l'algorithme de résolution que j'ai utilisé dans mon implémentation.

4.1.1 Le moment de traitement des différents type de contrats

Regardons maintenant à quel moment il est le plus opportun de valider l'ensemble des schémas.

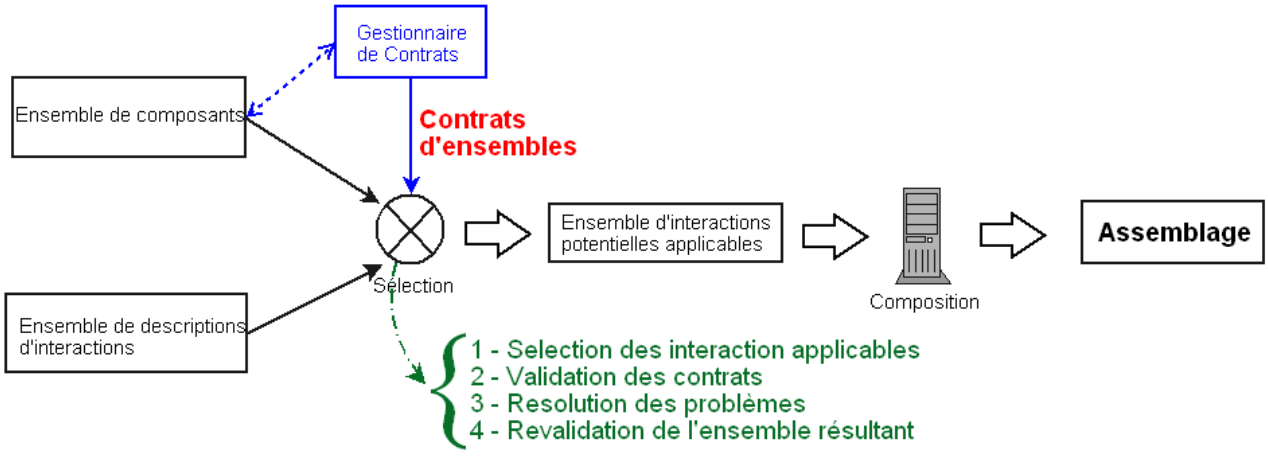

FIG. 4.1 – Validation des contrats sur les ensemble d'interactions potentielles

La solution est relativement évidente dans le cas des contrats portant sur un ensemble d'interactions potentielles. En effet le seul moment où l'on connaît l'ensemble des interactions potentielles pouvant intervenir est après la phase de selection, mais avant l'étape de composition. Le gestionnaire de contrat intervient bien à ce moment précis, comme l'indique la figure 4.1.

En ce qui concerne les contrats de type gardien, la réponse est bien moins évidente. Il y a deux étapes : la pose des gardiens, puis la réaction aux signaux qu'ils envoient. Cependant il semble intéressant que les différents type de contrat se valident de façon équivalente. C'est donc toujours le gestionnaire qui va sélectionner/désélectionner les interactions potentielles applicables après l'étape de selection préalable, mais avant la composition. La figure 4.2 illustre ce fait. Lorsque une sonde est activée, celle-ci récupère la valeur booléenne indiqué dans le contrat, puis l'envoie au gestionnaire de contrat. Celui-ci effectue les actions indiquée, puis refait une validation de l'ensemble des interactions potentielles sélectionnées.

L'ajout du gardien se fait lorsque le composant qui doit déclencher l'évaluation du contrat (c'est ce qui est mentionné après le mot clef "context" du contrat - cf partie 3.4 -) entre dans le système ; le retrait lorsqu'il en sort.

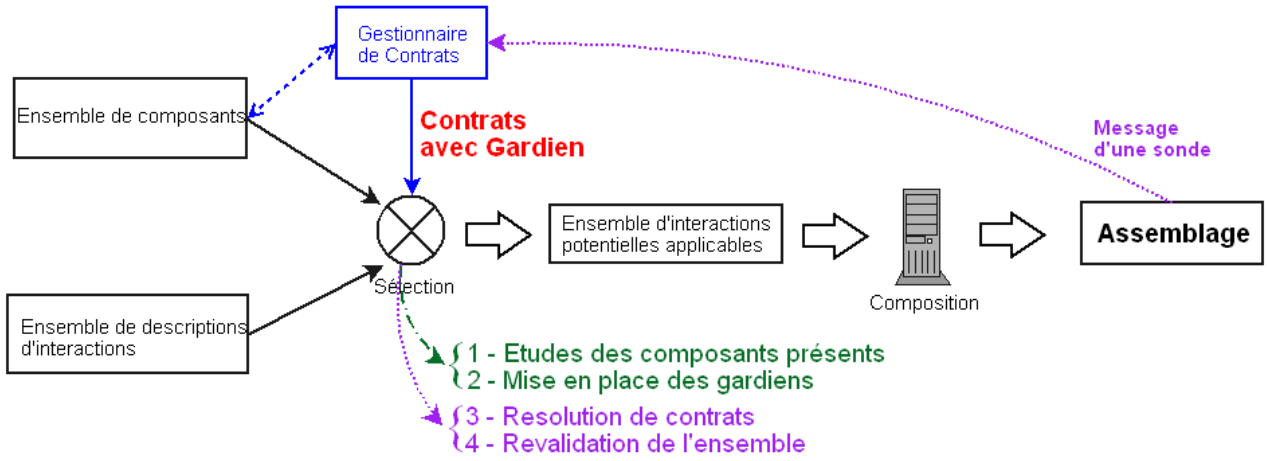

Fig. 4.2 – Validation des contrats avec gardien

4.1.2 Algorithme de composition et de résolution des contrats sur les ensembles

Je vais détailler ici l'algorithme que j'ai mis au point pour composer les contrats de type dépendance et exclusion mutuelle, ainsi que permettre d'extraire une solution d'un groupe d'interactions potentielles soumis à un ensemble de contrat d'exclusions mutuelles avec priorités.

Remarque : Pour cet algorithme, j'ai imposé qu'une solution, différente de l'ensemble vide, soit toujours trouvée, en signalant à l'utilisateur les hypothèses faites par l'algorithme. J'ai pris aussi pour parti de travailler par type de contrat, en groupant tout d'abord les interactions potentielles selon les contrats de dépendance pour y appliquer ensuite mes contrats d'exclusions mutuelles.

La figure 4.3 présente le déroulement de mon algorithme sur un exemple simple, dans le cas où nous ne travaillons qu'avec des exclusions mutuelles. Si ce n'est pas le cas, une première étapes consiste à regrouper les interactions potentielles selon les contrats de dépendance mutuelles, puis s'appliquer cet algorithme aux ensembles ainsi formés.

Le principe de cet algorithme est basé sur la construction d'un graphe orienté, les nœuds de ce graphe représentant les interactions potentielles {IP1, IP2,...,IP7} en exclusion mutuelle et les arcs symbolisant

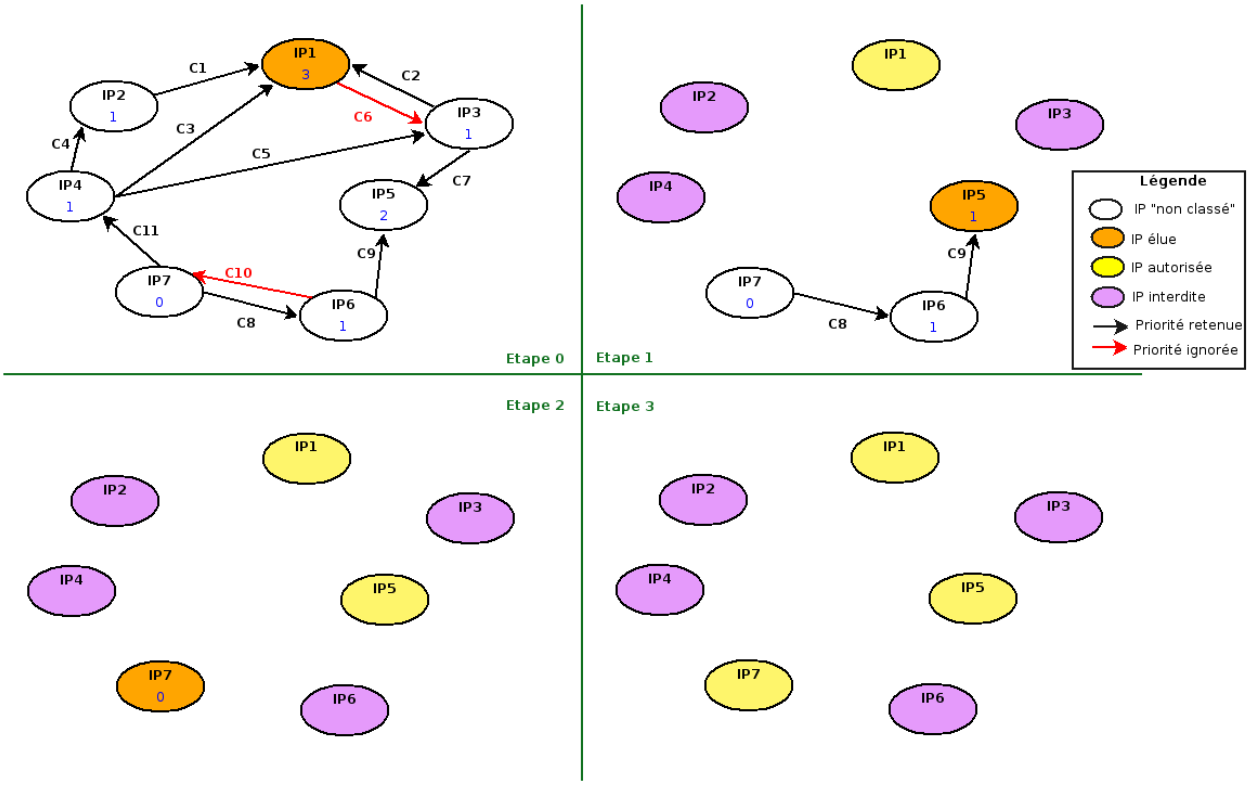

Fig. 4.3 – Application de l'algorithme de résolution sur un exemple

l'ordre de priorité indiqué par les contrats {C1, C2,...,C11}. Ainsi IP1 -> IP2 signifie que IP2>IP1, si le symbole ">" représente l'ordre de priorité entre les 2 éléments.

Les étapes de cet algorithme sont :

- Créer l'ensemble M des objets en exclusion Mutuelle (en blanc sur la figure 4.3).
- Créer le graphe orienté correspondant en vérifiant que l'on ne crée pas de cycle. Si une liaison allant d'un élément de priorité plus élevée, vers un élément de priorité moins élevé est proposée alors cet arc n'est pas créé et l'utilisateur est averti du problème. Sur la figure 4.3, les contrats en rouges (C6 et C10) vont être éliminé puisqu'en contradiction avec d'autres (respectivement C2 et C8) qui ont été "lu" avant par le système. Aujourd'hui, le choix de l'ordre de lecture des contrats est entièrement déterminé par le nom du contrat (ordre alphabétique strict).
- Élire le nœud ayant le plus d'arcs entrants (n). Sur la figure 4.3, le nombre d'arc entrant est noté en bleu en dessous du nom de l'interaction potentielle concernée, et l'élément élu est noté en orange. Si deux interactions potentielles ont le même nombre d'arcs entrants, celle sélectionnée sera celle qui aura le nom le plus petit selon l'ordre lexicographique.
- Déplacer l'élément n de M vers P (l'ensemble des éléments prioritaire, en jaune sur la figure 4.3), déplacer les fils de n (les éléments dont un arc sortant va en direction de n) vers S (l'ensemble des éléments Secondaire, en violet sur la figure 4.3).
- Recommencer les étapes précédentes avec M' = M \ {P,S} jusqu'à temps que cet ensemble soit vide

Revenons un instant sur le cas où l'on doit gérer simultanément des exclusions mutuelles et des dépendances.Le processus se fait en quatre étapes :
- Pour chaque interaction potentielle, on associe un nom d'ensemble
- On regroupe ces ensembles en fonction des contrats de dépendance
- On reformule l'ensemble des contrats d'exclusion afin de les adapter au nom d'ensemble. Ainsi si une interaction IP_1 appartient à l'ensemble E_1 et une interaction IP_2 à l'ensemble E_1, alors le contrat d'exclusion mutuelle $IP_1 > IP_2$ sera reformulé sous la forme $E_1 > E_2$.
  Remarque : Si jamais les deux interactions concernées par un contrat d'exclusion mutuelle appartiennent à un même ensemble, alors ledit contrat est ignoré et l'incohérence avec un contrat de dépendance est signalé à l'utilisateur.
- on applique l'algorithme précédent en utilisant pour sommet du graphe les ensembles définis à l'étape précédente.

4.2 Intégration à Wcomp

L'intégration dans Wcomp n'a posé que peut de problème car mon modèle trouve souvent son inspiration dans cette plateforme. Je vais cependant apporter quelques précisions sur l'implémentation du gestionnaire ISL4Wcomp dans cette plateforme ; avant de montrer rapidement mon apport, à travers un diagramme de composant et de quelques captures d'écran.

4.2.1 Description de l'intégration initiale d'ISL4Wcomp

Je vais détailler dans ce paragraphe le couple *Designer*/*Container* avec lequel j'ai travaillé durant mon stage. Je commencerais par exposer les différents niveaux (niveau statique et dynamique) Wcomp ; puis j'illustrerai brièvement mon implémentation.

Différents niveaux du designer ISL4WComp

Comme cité dans la partie 2.3, le fonctionnement de Wcomp est basé sur la complémentarité entre le *Container* et le *Designer*. Le programme géré par la plateforme Wcomp est constitué de composants Wcomp assemblé entre eux par le biais des interfaces serveurs et clients. En terme d'implémentation, ces liaisons ne sont que des événements qui sont émis par le composant source et traités par le composant destination. C'est le *Container* qui gère la mise en place des écouteurs d'événements. La figure 4.4 présente un exemple de programme tel qu'il est aperçu dans un *Designer* (ici celui intégré au logiciel open-source SharpDevelop[1])

Pour donner des perspectives de hiérarchisation de composant à la plateforme Wcomp, ainsi que pour montrer la réflexivité du modèle, il a été choisi que ce *Container* soit lui-même un composant Wcomp (tout comme le *Designer*). Le *Designer* et le *Container* sont relié entre eux par un bus logiciel (il a été utilisé plusieurs bus, en particulier Ice[2] et UPnP[3], mais d'autres, en particulier CORBA[4], pourraient être utilisés). Pour permettre de changer facilement le bus logiciel, ses fonctionnalités de communication (inter-plateforme) ont été isolé dans des composants Wcomp. Le composant principal et le composant de communication sont alors relié statiquement dans un executable.

La plateforme que nous utilisons possède donc bien deux niveaux :
- Un niveau dynamique où l'application développé s'exécute. Cette application est un assemblage de composants, assemblage réalisé par le *Container*.
- Un niveau statique composé du *Container*, du *Designer* et des différents composants leur permettant de fonctionner et de communiquer.

[1] `http://www.icsharpcode.net/OpenSource/SD/`
[2] `http://www.zeroc.com/ice.html`
[3] `http://www.upnp.org/`
[4] `http://www.corba.org/`

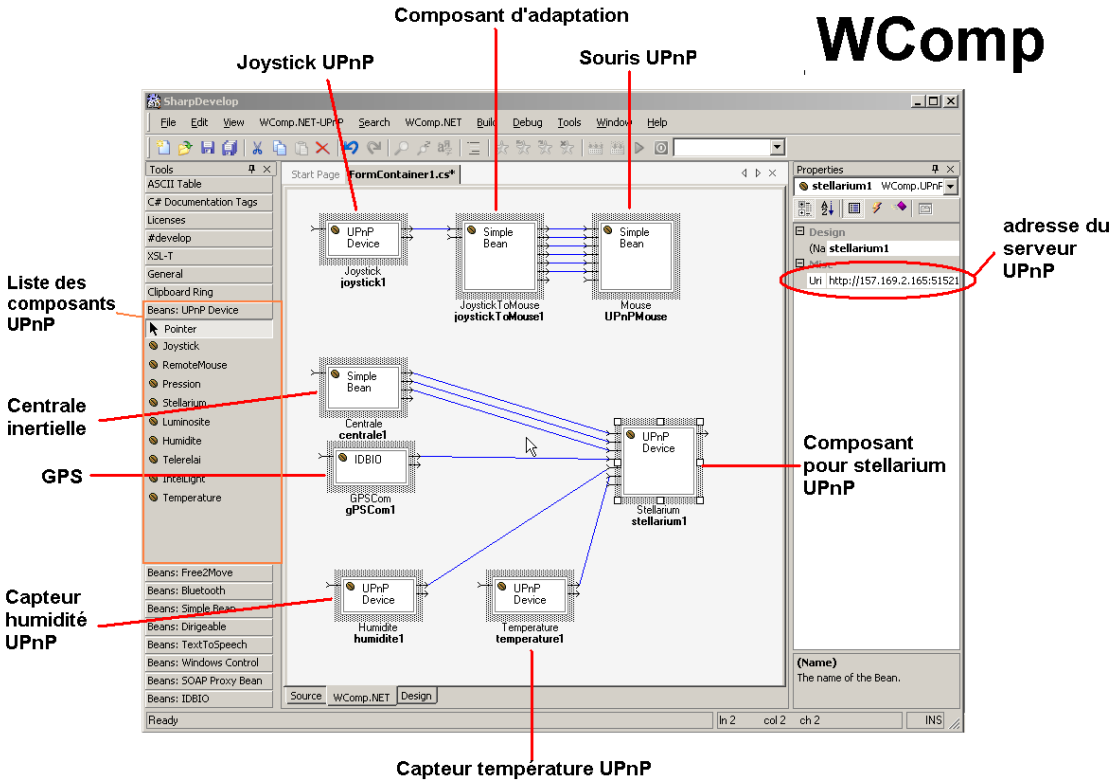

Fig. 4.4 – Exemple d'un programme Wcomp

Le programme qui est représenté ici est un programme de simulation de voûte celeste en temps réel(stellarium), couplé avec des composants permettant de se déplacer sur le globe et d'obtenir des informations sur les astres pointés. Il est aperçu ici avec un Designer intégré au logiciel open-source SharpDevelop

Une fois cette assertion acceptée, on pouvait déduire que différents types de contrats pouvaient exister et que ceux-ci pouvaient travailler soit sur le niveau statique, soit plutôt sur le niveau dynamique. Les contrats travaillant au niveau dynamique ont déjà été étudiés et ont déjà été mis en oeuvre avec ConFract. Mettre en place des contrats aussi complexe dans Wcomp ne semblaient pas possible puisqu'il manque une des notions fondamentale de Fractal à savoir la notion de hiérarchie (puisqu'un composant Fractal peut lui même contenir d'autres composants comme il a été signalé dans la partie 2.4.1). Cependant les contrats utilisant le niveau dynamique était intéressant pour satisfaire le troisième scénario d'utilisation défini dans la section 1.1.

Implémentation du *Designer* ISL4Wcomp

Pour permettre de comprendre, à posteriori, certains de mes choix, il est nécessaire de regarder d'un peu plus prêt l'implémentation dans son état initial, et plus précisément les étapes de fonctionnement du *Designer* ISL4Wcomp.

Comme nous le montre la figure 4.6, le processus qui permet de mettre en oeuvre les schémas d'Interaction ISL4Wcomp se déroule en quatres étapes :

– Lorsqu'une modification intervient dans le *Container*, celui signale qu'un changement a eu lieu. Par modification on entend l'ajout ou la suppression d'un composant ou d'une liaison. Il est notable que la

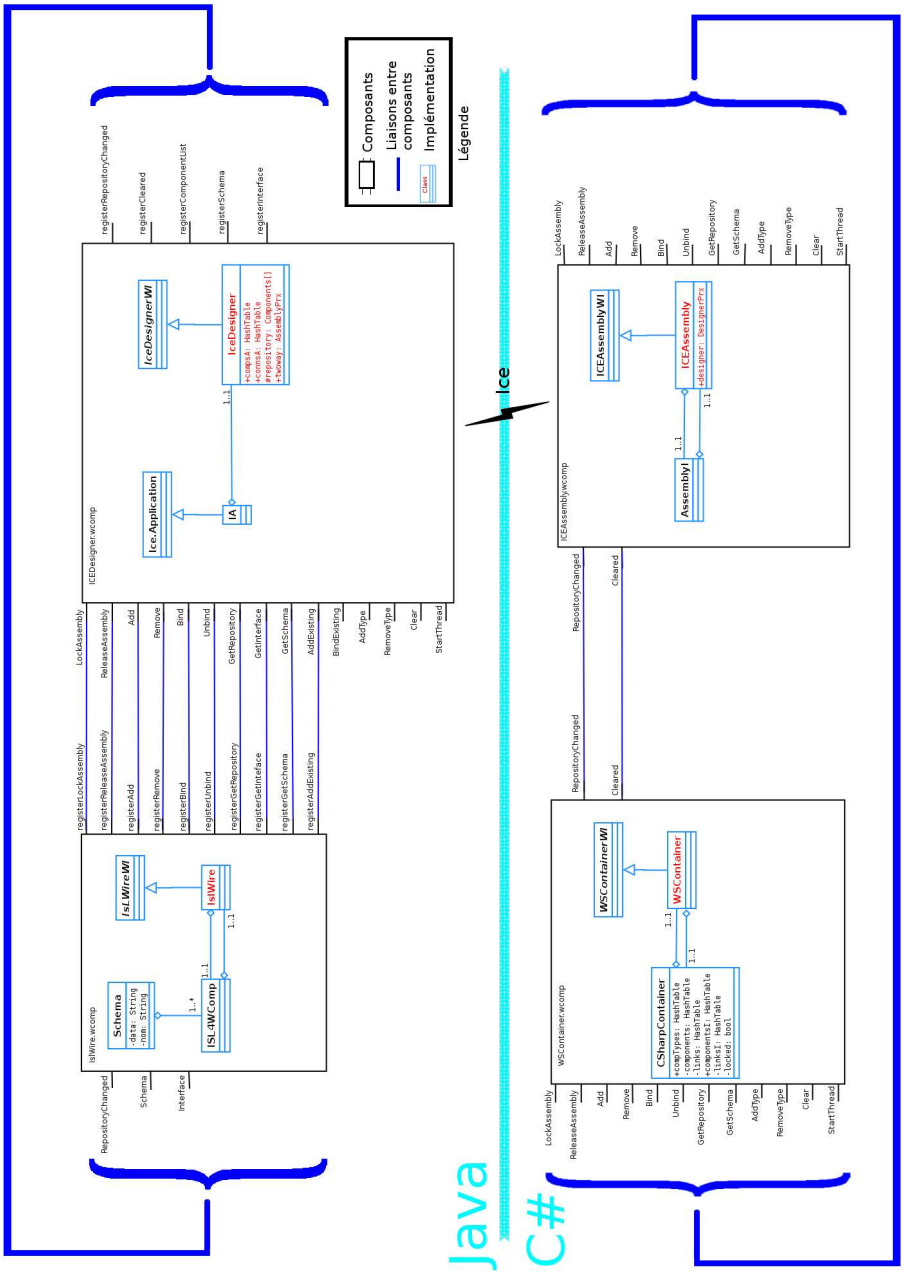

Fig. 4.5 – L'implémentation initiale du couple *Container* et *Designer* ISL4Wcomp utilisant le middleware ICE

modification qui engendre le déclenchement du processus provient d'un autre *Designer* et que donc le *Designer* ISL4Wcomp fonctionne en "partenariat" avec un autre.

– Lorsque le *Designer* ISL4Wcomp reçoit le signal comme quoi un changement dans le *Container* est intervenu, il fait une requête sur celui-ci afin que ce dernier lui communique son contenu. A partir de la liste des composants présents, les interactions potentielles applicables sont sélectionnées.
– Les Interactions Potentielles sélectionnées sont composées (en suivant un certain nombre de règles

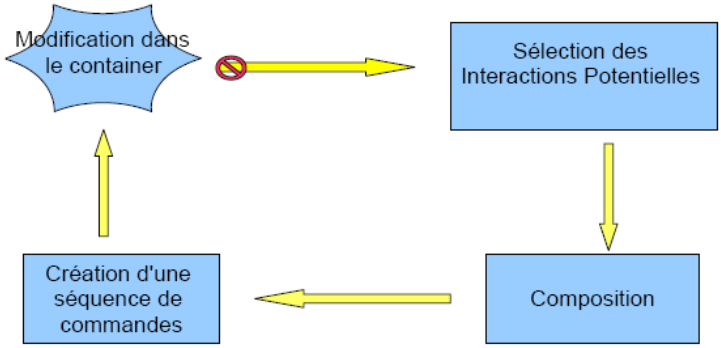

Fig. 4.6 – Description du processus de mise en place des schemas d'interaction ISL4Wcomp

définies dans le modèle ISL4Wcomp) c'est à dire que l'on extrait un assemblage qui valide l'ensemble des contraintes présentent dans les descriptions de liaisons. Cet assemblage est ensuite transformé en une suite de commandes primaires (Ajouter/Retirer composant, Créer/Détruire liens) qui vont être envoyé au *Container*.

– Le *Container* exécute la liste des commandes qui lui a été envoyée et signale que des changements sont survenu en son sein. (Remarque : Le sens interdit sur la flèche de la figure 4.6 symbolise un Sémaphore qui permet d'éviter que le *Designer* ISL4Wcomp réagisse aux changements qu'il a lui même engendrés).

4.2.2 Solutions d'intégration

Il est enfin temps d'illustrer mon travail au sein du laboratoire et mes apports sur la plateforme Wcomp.

Vue globale du système

La figure 4.7 montre une vue globale des différents composants qui ont été développés pour permettre le traitement des contrats.

Les nouveaux composants qui ont été introduit sont :

– le **selecteur** : il déporte le processus de selection des contrats applicables nécessaire au *Designer* ISL4Wcomp, représenté ici sous la dénomination *IslWire*. Ce sélecteur évolué connaît la liste interactions potentielles. Il embarque une interface graphique permettant d'afficher la liste des schémas sélectionnés.

– le **valideur de contrat** (contractValidator) : c'est lui qui connaît l'ensemble des contrats présents dans le système. Il possède un module de résolution de conflit entre contrats ainsi qu'une entrée permettant d'écouter les signaux provenant des gardiens contenus dans le *Container*. Il sélectionne et déselectionne les contrats applicables et donc, il assure que l'ensemble des interactions potentielles qui vont être sélectionnée par le sélecteur pour être composé par le module IslWire est cohérent avec l'ensemble des contrats posés par l'utilisateur. Il embarque également une interface graphique permettant d'afficher les contrats du système et de sélectionner ceux que l'on souhaite prendre en compte.

– le **générateur de chien de garde** (non présent sur la figure) : c'est un module qui travaille en collaboration étroite avec le *Container*. Son but est de créer à la volée des petits composants qui vont permettre de faire la liaison entre le gardien et le composant émettant le signal servant à déclencher l'évaluation des contrat avec gardien. L'introduction de ce composant est dû à une limitation du langage d'implementation du *Container* (C#). En effet, les signatures des méthodes du composants destination et les événements générés par le composant source doivent-être strictement identique.

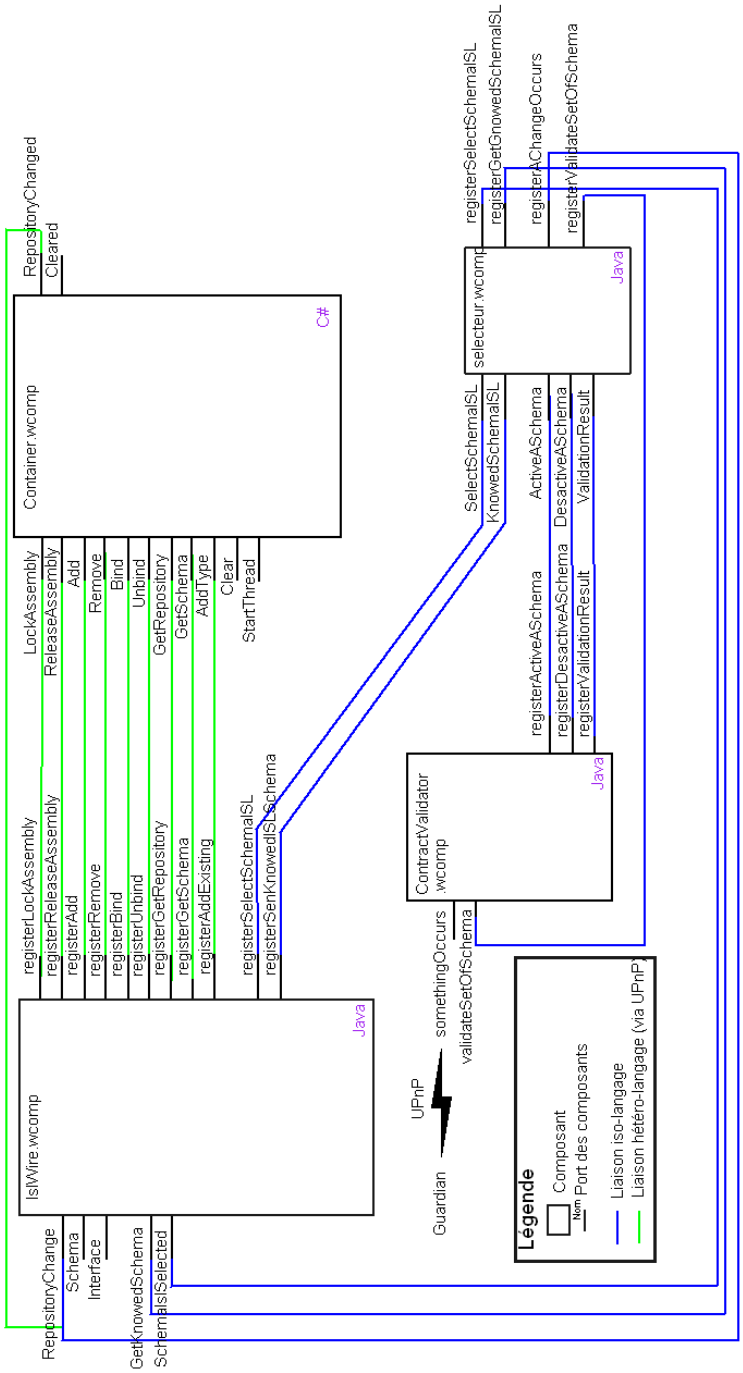

Fig. 4.7 – Assemblage statique des composants permettant le traitement de contrat sous Wcomp

Le problème majeur a été de faire communiquer les sondes avec notre valideur de contrat, afin de permettre une réaction aux événements du runtime. En effet, les composant Wcomp n'ont pas d'accès au *Container*. C'est ici qu'intervient le protocole UPnP. En effet, celui permet de relier un serveur et un point de contrôle indépendamment de toute hiérarchie.

Quelques captures d'écran

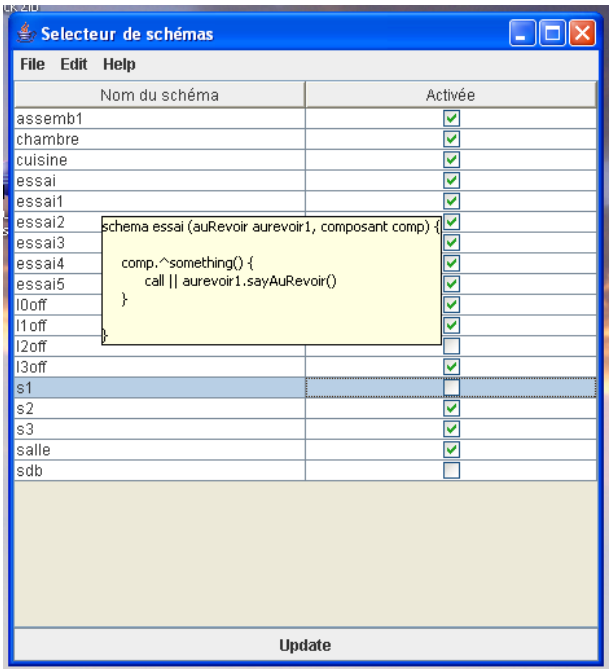

Fig. 4.8 – Fenêtre d'affichage des interactions potentielles connues du système avec leur état Autorisé/Interdit

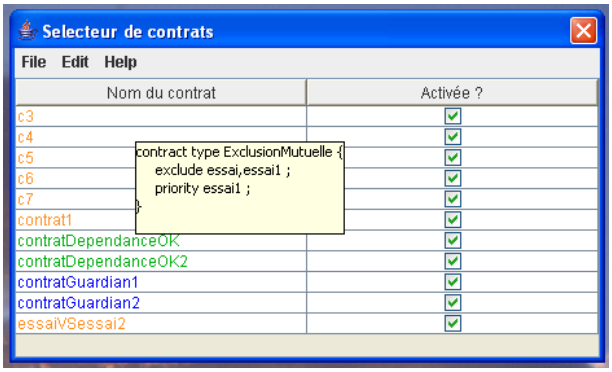

Fig. 4.9 – Fenêtre d'affichage des contrats connus du système(couleur selon le type) avec leur état Actif/Inactif

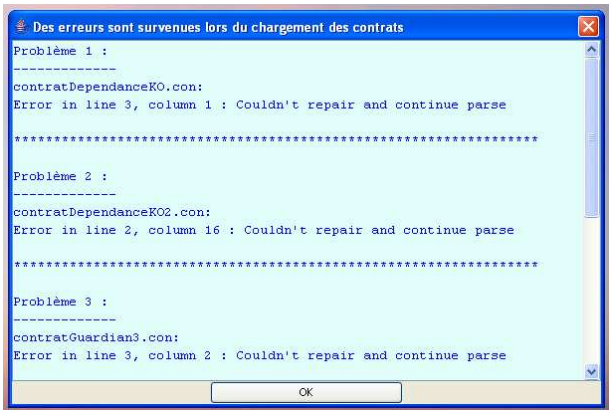

Fig. 4.10 – Fenêtre indiquant les erreurs lors du chargement des contrats (erreurs syntaxiques)

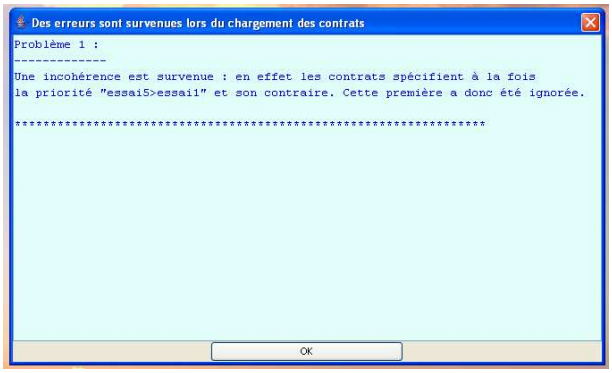

Fig. 4.11 – Fenêtre indiquant les erreurs lors de la validation des contrats (incohérences entre deux contrats)

Conclusion et perspectives

Au cours de ce stage de Master recherche, j'ai eu l'occasion de me familiariser avec de nombreuses technologies. Il m'a fallu appréhender la notion de composant, que nous avions entrevu durant notre cursus, mais dont l'intérêt ne nous était pas apparu clairement. J'ai aussi découvert le modèle d'interaction, développé par l'équipe, et sa souplesse d'utilisation. Enfin, j'ai pu tester les plateformes Fractal, ConFract et Wcomp.

Ces outils, que j'ai décrit dans mon état de l'art, m'ont permis d'établir le modèle global de la solution de mon problème : l'intégration de contrats afin de restreindre un ensemble d'interactions potentielles entre composants, et ainsi maîtriser mon système dans son adaptation à l'environnement.

L'implémentation, que j'ai effectué sur la plateforme Wcomp, résout les scénarios que l'on s'est fixé comme objectifs... Cependant, de nombreuses perspectives s'offrent à nous.

Vis à vis du modèle, un certain nombre de pistes restent à défricher :

- Les types de contrats que j'ai introduit sont-il applicable à d'autres plateformes, telle que ConFract, ou intégrable au modèle d'interaction et à NOAH afin de permettre de contrôler plus finement les assemblages de composants ?
- Peut-on enrichir le modèle afin de traiter des cas plus complexes ? Un cas d'utilisation pourrait-être celui-ci. Supposons un agenda A en relation avec un service de gestion S par une interface *IG1*. Supposons également un agenda d'équipe AE relié lui aussi au service de gestion par *IG1*. Le problème est de supprimer la liaison entre l'agenda A et l'interface *IG1*, et de la remplacer par une liaison avec l'interface *IG2*, uniquement dans le cas où l'on connecte l'agenda A à l'agenda d'équipe AE et que ce dernier est connecté sur le service de gestion S. La figure 4.12 illustre ce problème. En l'état, ce scénario ne peut pas être géré par notre modèle : il n'existe pas la notion d'opérateur ET et OU pour relier les différents contrats.

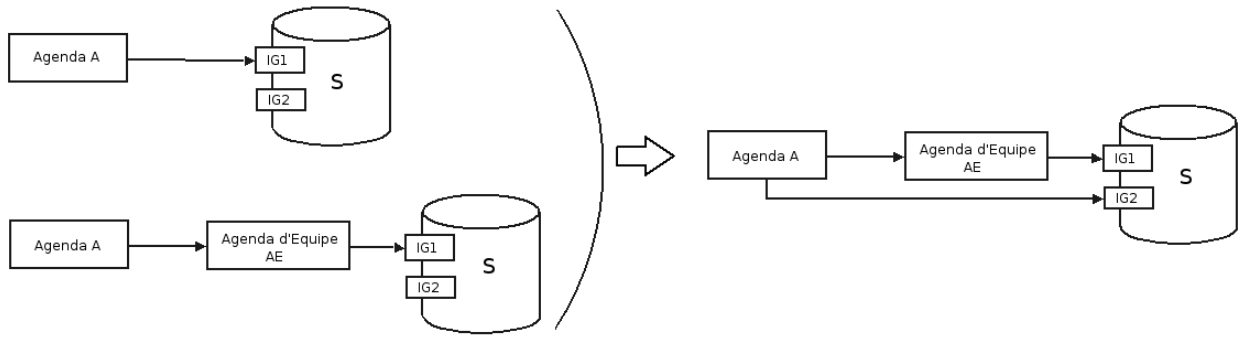

Fig. 4.12 – Un scénario non supporté par le modèle introduit

- Peut-on composer les différents types de contrat (contrats d'ensembles et contrats avec gardiens) ?

– Quelle est l'influence exacte de l'ordre de composition des contrats d'ensembles (exclusion mutuelle et dépendance) ?

De nombreuse améliorations sont aussi à apporter au niveau de l'implémentation :
– il faut vérifier s'il est plus pertinent de travailler chaque type de contrat indépendamment.
– il faut améliorer l'algorithme de composition qui a été mis en place pour permettre le choix de l'ordre entre le traitement des contrats de dépendances et ceux d'exclusions mutuelles.

Sur un plan plus personnel, ce stage m'a permis de découvrir le monde de la recherche, ses joies et ses frustrations les jours creux. Il m'a aussi permis de me familiariser avec un mode de programmation qui va se démocratiser puisqu'elle permettra à terme un développement plus rapide et donc moins coûteux.

Bibliographie

[1] Component object model technologies. http ://www.microsoft.com/com/default.mspx.

[2] Corba component model. http ://ditec.um.es/∼ dsevilla/ccm/index.shtml.

[3] Enterprise java beans. http ://java.sun.com/products/ejb/.

[4] Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valerie Issarny. Towards efficient matching of semantic web service capabilities. In *WS-MaTe - International Workshop on Web Services - Modeling and Testing*, Palermo (Italy), June 2006.

[5] Laurent Berger. *Mise en oeuvre des intéractions en environnements distribués, compilés et fortement typés : le modèle MICADO.* PhD thesis, Université de Spohia Antipolis, October 2001.

[6] Mireille Blay-Fornarino, Anis Charfi, David Emsellem, Anne-Marie Pinna-Déry, and Michel Riveill. Software interaction. *Journal of Object Technology (ETH Zurich)*, 3(10) :161–180, 2004. http ://www.jot.fm/issues/issue_2004_11/article4.

[7] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stephani. Recursive and dynamic software composition with sharing. *Processings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, June 2002.

[8] Hervé Chang. Mécanisme de négociation pour composants logiciels contractualisés. Rapport de stage, laboratoire I3S, Avril–Juin 2004.

[9] Anis Charfi, David Emsellem, and Michel Riveill. Dynamic component composition in .Net. *Journal of Object Technology (ETH Zurich)*, 3(2) :37–46, February 2004. http ://www.jot.fm/issues/issue_2004_02/article4.

[10] Daniel Cheung Foo Wo, Mireille Blay-Fornarino, Jean-Yves Tigli, Anne-Marie Pinna-Déry, David Emsellem, and Michel Riveill. Langage d'aspects pour la composition dynamique de composants embarqués. 2005.

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

[12] Philippe Collet and Alain Ozanne. Un système de contractualisation pour fractal : Intégration et retours sur expérience. Rapport de recherche ISRN I3S/RR-2005-01-FR, Laboratoire I3S, Projet OCL, January 2005.

[13] Philippe Collet and Roger Rousseau. Confract : Un système pour constractualiser des composants logiciels hiérarchiques. Rapport de recherche ISRN I3S/RR-2004-32-FR, Laboratoire I3S, Projet OCL, October 2004. http ://www.i3s.unice.fr/∼mh/RR/2004/RR-04.32-P.COLLET.pdf.

[14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object–Oriented Programming(ECOOP)*, Finland, June 1997.

[15] Tom Mens, Günter Kniesel, and Olga Runge. Transformation dependency analysis. A comparison of two approaches. In *Langages et Modèles à Objets (LMO)*, pages 167–182, Nimes, March 2006. Hermes.

[16] Bertrand Meyer. Eiffel : Le langage. In *Object-Oriented Series*, Prentice Hall, New York, NY, June 1992.

[17] Clémentine Nemo. Vers la composition d'orchestrations de services. Master dissertation, DEA PLMT, Nice (France), June 2006.

[18] Clémentine Nemo, Mireille Blay-Fornarino, and David Emsellem. Composition d'orchestrations de services. In *Atelier sur l'Evolution du Logiciel*, pages 53–60, Nimes, March 2006. Salah Sadou.

[19] Oana Novacescu. Des IHMs composables pour les applications à base de composants. Master's thesis, Université de Nice - Sophia Antipolis, Sophia Antipolis (France), June 2006. http ://-diane.essi.fr/publis/masterOanaNovacescu2006.pdf.

[20] Anne-Marie Pinna-Déry and Jérémy Fierstone. Construction d'Interfaces Utilisateurs Par Fusion de Composants d'IHM : un Atout Pour la Mobilité. 2004. http ://www.springeronline.com/sgw/cda/frontpage/0,10735,5-164-22-7131582-0,00.html.

[21] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4) :314–335, 1995. http ://dx.doi.org/10.1109/32.385970.

[22] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. Jml : notations and tools supporting detailed design in java. In *OOPSLA '00 Companion*, pages 105–106, Minneapolis, Minnesota, 2000. ftp ://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.pdf.

[23] Benoît Vallette d'Osia. Traitement des exceptions dans des composants hiérarchiques autonomes. Rapport de stage, laboratoire I3S, Mars–Août 2005.