
Langage d'aspects pour la composition dynamique de composants embarqués

**Daniel Cheung-Foo-Wo, Mireille Blay-Fornarino,
Jean-Yves Tigli, Anne-Marie Dery, David Emsellem
et Michel Riveill.**

*Université de Nice – Sophia Antipolis
Laboratoire I3S CNRS UMR 6070
930 Route des colles, 06903 Sophia Antipolis Cedex
{cheung,blay,tigli,pinna,emsellem,riveill}@essi.fr*

RÉSUMÉ. Bien que l'on observe la maturation des supports système destinés à simplifier le développement de systèmes embarqués, on constate leur inadéquation dans le cadre d'environnements où les équipements peuvent apparaître ou disparaître spontanément. En se basant sur des applications sous forme d'assemblage de composants, l'adaptation se traduit par une modification dynamique de leur assemblage. Nous proposons de tisser des modèles d'assemblage selon l'apparition ou la disparition spontanée de certains composants et en fonction des besoins de l'utilisateur et du contexte d'usage. Cette méthode facilite l'expression de l'évolutivité de ces applications à un niveau déclaratif. Nous avons alors défini un langage d'aspect d'assemblage que nous composons dans la plate-forme Wcomp. Le code ainsi généré s'adapte aux besoins des applications embarquées.

ABSTRACT. Although supports are matured to simplify embedded system development, they are not yet adapted to environments where equipments can appear or disappear. If the application is based on components, its adaptation will result in a dynamic modification of their assembly. We propose weaving assembly models according to the components presence and in function of the user needs and the context of use. This method makes the expressiveness of the evolution of those applications easy at a declarative level. We have defined an assembly-aspect language built upon the Wcomp framework. The generated code is then adapted to the needs of embedded applications.

MOTS-CLÉS: Systèmes embarqués, Aspect, Assemblage, Composant.

KEYWORDS: Embedded Systems, Aspect, Assembly, Component.

1. Introduction

Avec l'avènement des systèmes mobiles et ubiquitaires, les logiciels nécessitent toujours plus d'adaptation face aux modifications de leur contexte d'usage. Ils s'adaptent notamment aux variations de l'environnement (qui peut être au départ bruyant puis devenir silencieux), aux contextes d'usage (si l'utilisateur est assis, debout, à l'arrêt ou en train de marcher) et aux équipements communicants disponibles (des prises électriques ou des lampes communicantes).

Le paradigme qui permet de gérer une application logicielle par assemblage de composants s'avère très pertinent car l'adaptation d'une application se traduit par une modification dynamique des assemblages de composants qui la constituent. Les concepteurs de logiciels pour les systèmes mobiles ont les préoccupations suivantes : économiser l'énergie et connecter les applications logicielles. Pour prendre en charge les adaptations, les concepteurs expriment en fonction de ces préoccupations des modèles cohérents d'assemblage en présence d'équipements communicants.

Notre objectif est d'appliquer et de *tisser* dynamiquement ces modèles d'assemblage en fonction d'une part de l'apparition ou de la disparition de certains équipements communicants et d'autre part des besoins des usagers et du contexte d'usage. Les assemblages de composants évoluent ainsi au grès de ce contexte. Cet objectif nous a conduits à nous intéresser à l'approche par aspects pour assembler nos composants embarqués dans la plate-forme Wcomp [6]. Nous avons choisi de définir un *langage d'aspects d'assemblage* : à partir de l'expression des différents *aspects d'assemblage* et en fonction d'un ensemble donné de composants logiciels, nous déterminons l'assemblage correspondant. Nous projetons dans la plate-forme le résultat obtenu en termes de connexions et d'instanciation de composants existants et de composants générés pour les besoins.

Le travail décrit dans cet article s'appuie sur une application développée dans le cadre d'une collaboration entre le laboratoire I3S¹ et le CSTB². La section 2 présente succinctement cette application et la plate-forme ciblée. Dans la section 3, nous présentons le langage d'aspects ISL/Wcomp, extension du langage ISL [2]. Cette extension a été définie pour simplifier l'expression des modèles d'assemblage (cf. 3.1). Nous montrons comment le tissage nous permet de composer ces assemblages. Nous décrivons ensuite les assemblages de composants obtenus au niveau de la plate-forme cible Wcomp (cf. 3.2). Enfin nous montrons comment l'approche nous permet de prendre en charge la composition dynamique des aspects d'assemblage (cf. 3.3). Nous concluons cet article par les perspectives de ce travail (cf. 4).

1. Informatique Signaux et Systèmes de Sophia Antipolis

2. Centre Scientifique et Technique des Bâtiments de Sophia-Antipolis

2. Application cible

Nous nous intéressons, dans cette application fournie par le CSTB, à la connexion de composants de manière dynamique. L'objectif est d'établir ces connexions entre composants en fonction de modèles d'assemblage préétablis par différents concepteurs et en fonction des choix des usagers. Ces modèles sont exprimés séparément en fonction de la nature des composants et des objectifs.

Nous avons dégagé les besoins suivants :

- Un langage de haut niveau pour exprimer les modèles d'assemblage entre les composants.
- Ce langage doit permettre de contrôler les émissions d'évènements et la réception de messages.
- Ces modèles doivent pouvoir être exprimés séparément sans se préoccuper des autres modèles.
- Différents modèles doivent pouvoir être appliqués simultanément sur des ensembles de composants. Il faut alors tisser les contrôles qu'ils imposent.
- Il est nécessaire de contrôler la validité des assemblages obtenus.

Pour toutes ces raisons, le langage à définir doit reposer sur la programmation et la modélisation par aspects.

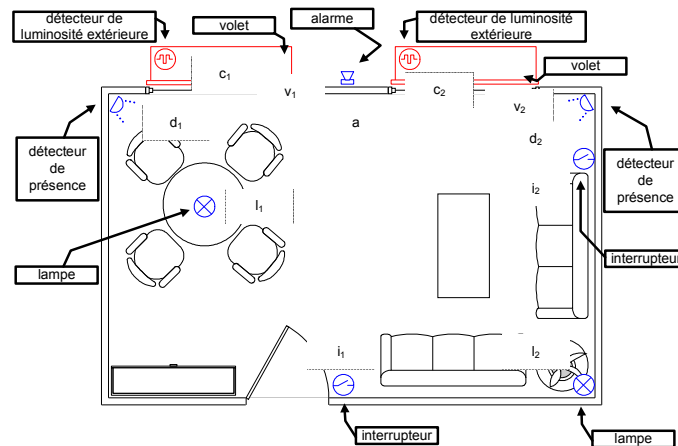


Figure 1 : Pièce d'habitation de type séjour/salon

2.1. Application à un bâtiment haute technologie

Le système applicatif ciblé initialement est constitué d'une pièce dans laquelle on positionne successivement différents équipements. Ces équipements sont :

- Des interrupteurs qui envoient un évènement nommé $\wedge\mathbf{on}()$ et un évènement $\wedge\mathbf{off}()$,
- Des lampes qui peuvent recevoir des messages sur une opération $\mathbf{on}()$ et une opération $\mathbf{off}()$,

- Des détecteurs de lumière positionnés à l'extérieur de la pièce qui envoient spontanément toutes les N secondes un évènement `^lightValue(integer value)` contenant une information sur l'intensité lumineuse à travers le paramètre `value`,
- Des détecteurs de présence qui émettent soit un évènement `^nobody()`, soit un évènement `^somebody()` lorsque l'opération `scan()` est appelée.

Ils sont représentés au niveau logiciel par des types précis de composants. Ces composants communiquent par *réception de message* et *émission d'évènements*. Différentes connexions doivent être établies en fonction de la nature des composants présents et des modèles d'assemblage choisis.

Notre objectif est de maintenir la *cohérence* de l'application dans son ensemble tout en autorisant la définition séparée des modèles d'assemblage et l'évolution dynamique de l'application.

Expression d'aspects

Différents modèles d'assemblage peuvent être définis. Ces modèles doivent être définis séparément pour permettre des compositions distinctes en fonction du contexte d'usage [15]. Ces modèles expriment différents contrôles sur les communications entre composants. C'est pourquoi nous nommons à partir de maintenant ces modèles, aspects. Voici quelques exemples :

- Cet exemple traite des connexions entre composants. Il s'agit de décrire par des aspects la réaction du système à l'émission de certains évènements.

Les aspects qui connectent deux composants l'un jouant le rôle d'interrupteur et l'autre de lampe peuvent être décrits de manière littérale par :

n°	Nom du schéma	Description
1	switch_light_on	Dès que l'interrupteur émet l'évènement <code>^on</code> , alors la lampe s'allume.
2	switch_light_off	Dès que l'interrupteur émet l'évènement <code>^off</code> , alors la lampe s'éteint.

L'aspect qui connecte un détecteur de présence à une alarme sera simplement décrit par :

n°	Nom du schéma	Description
3	alarm	Si le détecteur émet l'évènement <code>^somebody</code> lorsque <code>scan</code> est appelée, alors déclencher l'alarme.

Si l'alarme est éteinte, cet aspect n'aura aucune conséquence.

- Certains assemblages visent à conditionner la propagation d'évènements. Ils se traduisent par des aspects qui contrôlent la réception de messages ou l'émission d'évènements.

L'aspect qui connecte un détecteur de présence à une lampe dans une pièce se décrit par :

n°	Nom du schéma	Description
4	light_and_detector	Si le détecteur émet l'évènement <code>^nobody</code> (lorsque <code>scan</code> est appelée), éteindre la lampe. Et la lumière ne doit être allumée que si une personne est présente.

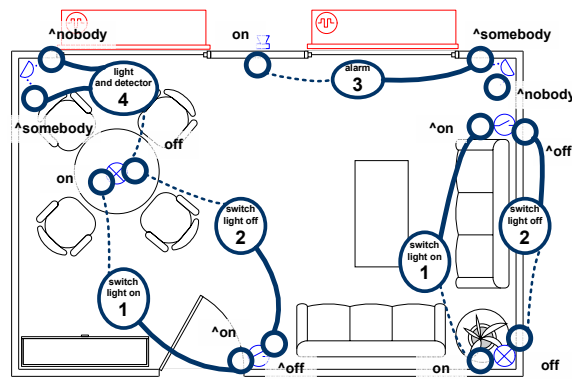
Une autre préoccupation peut porter sur l'économie d'énergies. Au lieu d'allumer la lumière, nous voulons que les volets s'ouvrent lorsqu'il y a suffisamment de lumière extérieure.

n°	Nom du schéma	Description
5	light_control	La lampe ne doit être allumée que s'il n'y a pas suffisamment de luminosité à l'extérieur sinon, nous devons ouvrir les volets.
6	help_user	Dès que la luminosité extérieure est insuffisante, allumer la lampe et fermer les volets.
7	delay	Il est parfois nécessaire dans certains cas de temporiser l'émission d'évènements.

Nous avons donc défini sept schémas d'assemblage que nous allons composer. Voyons à présent comment s'opère le passage entre la composition de ces schémas et l'assemblage des composants.

Connexion de composants et « application » d'aspects

A partir des aspects définis précédemment, il s'agit de construire des assemblages cohérents de composants. Soit l'application constituée des composants suivants : deux interrupteurs, deux lampes, deux détecteurs de présence, deux volets, deux capteurs de luminosité extérieure et une base de temps que nous allons appeler *timer*.



Chaque interrupteur allume et éteint une lampe donnée par l'application des deux schémas **switch_light_on** (1) et **switch_light_off** (2). Lorsqu'il n'y a plus personne dans la zone détectée par le détecteur de présence, la lampe qui lui est associée est alors éteinte (cf. **light_and_detector** (4)).
Finalement, la détection d'une présence dans le salon déclenche l'alarme (cf. **alarm** (3)).

Figure 2 : Aspects et composants

Tissage d'aspects sans fusion

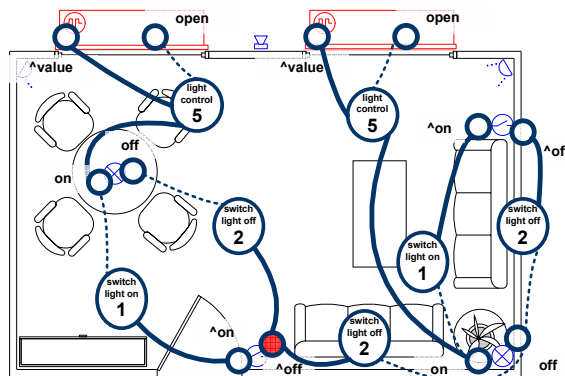
La Figure 2 présente une configuration possible prenant pour base les schémas 1, 2, 3 et 4. Dans cette configuration, tous les aspects s'appliquent sur des points de jonction disjoints, on ne trouve aucun cas de fusion. Une fusion est un cas particulier de la composition qui fait intervenir plus qu'une simple juxtaposition de règles disjointes.

Dans ces figures, les **traits en gras** symbolisent l'application d'une règle sur une opération ou sur un évènement. Les **traits fin** en pointillés symbolisent l'utilisation d'une opération. Les **petits cercles** sont soit des évènements (lorsque leur nom est préfixé d'un « ^ »), soit une opération. Ces cercles se rattachent à un équipement : par exemple, en haut à gauche de la Figure 2, les cercles **^nobody** et **^somebody** se rattachent au détecteur de présence (demi-cercle bleu sur le dessin).

Tissage d'aspects avec une fusion

Avec la même base de composants, nous pouvons définir une autre configuration relative à un autre usage des composants physiques telle que celle décrite par la Figure 3. Celle-ci met en jeu du tissage (une fusion) entre aspects.

En effet, lorsque l'interrupteur du hall émet l'évènement **^off**, la fusion des deux schémas n°2 induit le scénario suivant : actionner **^off** sur l'interrupteur amène l'extinction des deux lampes.



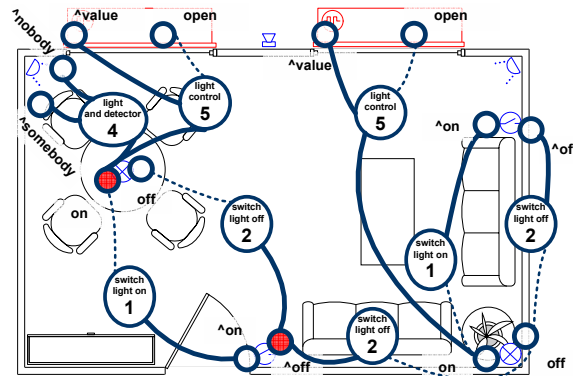
L'interrupteur situé près de la porte allume et éteint la lampe de la salle à manger. Il éteint également la lampe du salon. Les capteurs de luminosité contrôlent quant à eux l'éclairage des salles à proximité.

Figure 3 : Premier exemple de tissage avec fusion de schémas d'assemblage

Tissage d'aspects avec deux fusions

A l'émission de certains évènements peuvent correspondre *des réactions composées*. Par exemple dans la Figure 3, lorsque l'interrupteur du hall émet l'évènement **^off**, par application de deux aspects n°2, il déclenche l'extinction des deux lampes dans un ordre non déterminé. L'assemblage des composants par les aspects génère une *propagation des réactions*. Dans la Figure 3, appuyer sur l'interrupteur de l'entrée doit allumer la lampe selon le schéma n°1. Toutefois selon le schéma n°5, la lumière ne doit être allumée que s'il n'y a pas suffisamment de lumière extérieure. Ainsi appuyer sur l'interrupteur pourra avoir pour conséquence d'ouvrir les volets. Il s'agit dans ce cas d'une *propagation*. Le tissage des aspects est nécessaire lorsqu'un même point de jonction est contrôlé par plusieurs aspects.

Dans la Figure 4, nous avons d'abord le schéma n°4 qui stipule qu'il ne faut allumer la lampe que si une personne est détectée dans la pièce. Puis, le schéma n°5 substitue à l'éclairage de la lampe l'ouverture des volets lorsqu'il y a suffisamment de luminosité extérieure.



L'éclairage de la lampe de la salle à manger est conditionné par la détection d'une personne dans cette pièce et l'insuffisance de luminosité extérieure.

Figure 4 : Deuxième exemple de tissage avec fusion de schémas d'assemblage

Il y a alors fusion de deux schémas d'assemblage. En effet, si la lumière extérieure est suffisante et qu'une personne est présente dans la pièce, alors s'ouvrent les volets. Et inversement, les volets ne s'ouvriront, en réponse à une demande d'allumage, que si une personne est présente.

Evolution dynamique de l'assemblage de composants

L'ajout ou le retrait d'aspects est ici orchestré par des stratégies de plus haut niveau telles que la reconnaissance d'un contexte (présence d'un certain usager), la mise en place de politiques de consommation différentes comme le confort (un même interrupteur pour éteindre toutes les lampes) ou le coût (favoriser l'utilisation de la lumière extérieure). Nous n'en discuterons pas dans cet article pour nous focaliser sur les conséquences techniques de l'adaptation du système.

Le retrait d'un composant implique la destruction des aspects dans lesquels il était impliqué. Par exemple, dans la configuration donnée en Figure 3, supprimer le détecteur de présence implique le retrait de l'aspect n°4 de notre réseau de composants. Ajouter une nouvelle lampe peut impliquer d'appliquer un nouvel aspect entre l'interrupteur de l'entrée et cette lampe et donc une composition des aspects existants.

2.2. Plate-forme ciblée

Wcomp est une plate-forme logicielle/matérielle basée sur un modèle de composants dynamiques pour les systèmes embarqués dotés de réseaux de capteurs

ou d'actionneurs. Un composant logiciel dans Wcomp est l'instance d'une classe. Un composant est éventuellement muni d'une *interface serveur* qui est un ensemble d'opérations et/ou d'une *interface client* qui se compose d'évènements qui peuvent être émis. Un évènement transmet à la fois un fil d'exécution et des données. On distingue deux catégories de composants Wcomp : les composants logiciels (appelés simplement *composants*) et les composants Mixtes c'est-à-dire un composant logiciel représentant un équipement matériel. Nous expliciterons ce qu'est un composant Mixte dans la partie *Assemblage* à travers quelques exemples.

Nous allons d'abord expliquer succinctement le mécanisme de découverte des équipements, puis le cycle de vie des composants dans la plate-forme et enfin donner des exemples d'assemblage de composants logiciels ou Mixtes en relation avec les aspects d'assemblage décrits précédemment.

Découverte des équipements et instanciation des composants. La découverte automatique de composant se fait par un contrôleur logiciel du média de communication. Il scrute de nouveaux équipements à intervalle régulier et en fonction de leur présence et de leur type, il instancie les composants logiciels adéquats.

Cycle de vie. L'assemblage est modifié par ajout ou retrait de composants ou de connexions. La tâche s'occupant de la modification des assemblages préempte les autres tâches dites fonctionnelles. D'autres techniques plus précises peuvent être utilisées comme dans [21], consistant à optimiser l'ordonnancement des tâches en tenant compte de la reconfiguration matérielle. Nous ne le détaillerons pas dans cet article.

Un composant qui est retiré est immédiatement arrêté. Un composant instancié est immédiatement démarré (s'il est actif). Comparé à d'autres plates-formes à composants pour les systèmes embarqués et des réseaux de capteurs comme [9], les composants Wcomp fonctionnent sur une machine virtuelle embarquée. L'arrêt d'un composant consiste à libérer la mémoire allouée par l'instance du composant. Cette libération est assurée par un *ramasse-miettes*. La gestion du cycle de vie est réalisée par les chargeurs dynamiques de classes [10][14] aidés par des mécanismes de *réflexion* inspirés des travaux de [19]. Wcomp contribue dans le domaine des systèmes embarqués pour y porter les mécanismes de reconfiguration dynamique comme dans DYVA [12][13][14] (non embarqué) ou partiellement (pas tous les composants peuvent être dynamiquement ajoutés, retirés ou connectés) comme dans Think [4].

Nous ne nous intéressons pas dans cet article à la mise à jour de composants qui posent des problèmes liés à la compatibilité d'interface : création de composants d'adaptation d'interface. Les composants qui dépendent de cette mise à jour doivent également être considérés à cause, par exemple, de l'incompatibilité des nouvelles structures de données introduites. Les travaux comme ACCEEL [5] et Chisel [11] (successeurs de DART [19]) proposent des méthodes et des supports pour ces

problèmes. D'autre part, pour créer une application, des connexions entre ces composants doivent être établies, on dit alors qu'on les assemble.

Composants et assemblage. 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 événements. Ils sont notifiés via leur interface serveur par une transformation adéquate de l'évènement en un envoi de message prise en charge 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 nouveau fil d'exécution c'est-à-dire de *thread* ou de tâche, et actifs dans le cas contraire.

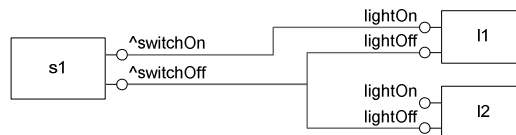


Figure 5 : Représentation graphique d'un assemblage de Wcomp

Par exemple, soit *s1* un interrupteur qui présente une interface client composée de **^switchOn** et de **^switchOff**. *l1* et *l2*, deux lampes qui présentent une interface serveur **lightOn** et **lightOff**. La connexion de *s1* par l'évènement **^switchOn** à *l1* et la connexion des lampes *l1* et *l2* à l'évènement **^switchOff** sont représentées dans la **Figure 5**. Dans cet exemple, *s1* est un composant actif et *l1* et *l2* sont passifs. *s1*, *l1* et *l2* sont ces fameux composants Wcomp Mixtes : ce sont des composants logiciels composés d'une partie matérielle : l'équipement. Les équipements peuvent eux aussi être actifs (émettre des informations sur un certain média de communication) ou passifs (en attente d'un ordre pour éventuellement fournir une information). Par conséquence, la partie logique du composant Wcomp Mixte est de même nature.

Introduction de contrôles dans les assemblages. Le contrôle de l'exécution d'une opération en fonction des valeurs d'un autre composant s'exprime par l'introduction dans l'assemblage d'un composant contrôleur.

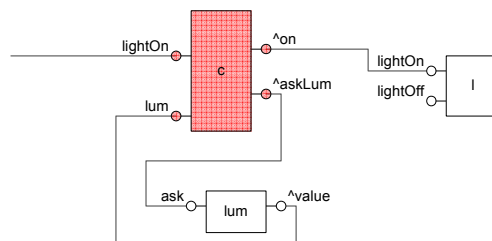


Figure 6 : Capteur et contrôle de l'allumage

Exemple : Moduler l'allumage d'une lampe en fonction de la valeur d'un capteur de luminosité

Un évènement $\wedge\text{ask}$ est émis par le contrôleur c . Le capteur lum est connecté au contrôleur. Il répond à cet évènement par l'émission d'un évènement $\wedge\text{value}$. Le contrôle d'évènements consiste à introduire un contrôleur au niveau de l'interface cliente du composant à contrôler.

Exemple : Emission différée de l'évènement signalant une absence de présence

L'objectif est de ne pas déclencher les composants connectés dès qu'il n'y a plus personne dans la pièce, mais d'attendre que la personne soit vraiment sortie. Il s'agit d'ajouter un temporisateur sur l'évènement $\wedge\text{nobody}$ émis par le détecteur de présence d . L'algorithme du $timer$ consiste à différer l'envoi d'un évènement. Si un autre évènement survient au moment où le $timer$ a déjà été déclenché, celui-ci est ignoré. L'évènement différée est $\wedge\text{handler}$.

Représentants logiciels des capteurs actifs : capteurs passifs. Un composant Wcomp Mixte actif peut devenir passif grâce à un composant Wcomp logiciel supplémentaire de *bufferisation* et qu'un composant Wcomp Mixte passif peut devenir actif s'il est piloté par un composant logiciel actif.

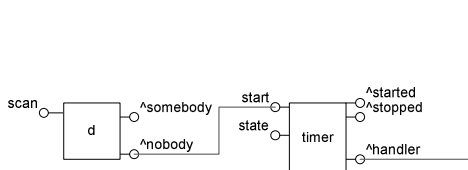


Figure 7 : Contrôle différé

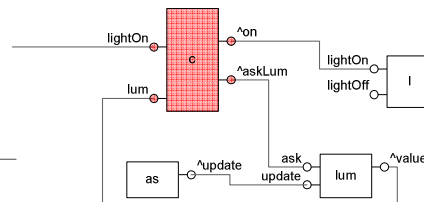


Figure 8 : Capteur actif et capteur passif pour le contrôle de l'allumage

Pour illustrer la *passivation*, nous vous proposons l'exemple suivant qui rend passif un certain équipement : capteur de luminosité actif. Cette transformation logicielle est générée automatiquement par l'introduction d'un composant logiciel supplémentaire lum qui contient l'état du capteur actif as . Cela nous permet de traiter les composants Mixtes comme étant tous passifs : le code des aspects est alors simplifié.

3. Assemblage des composants par aspects

Nous avons choisi de répondre aux besoins de cette application, en développant un langage d'aspect dédié. Les aspects sont décrits en respectant les interfaces des composants Wcomp. La composition des aspects est réifiée au niveau d'un modèle de l'application. Les calculs de tissage des aspects et l'évolution dynamique des assemblages sont actuellement pris en charge au niveau de ce modèle (cf. 3.1). Le résultat de ces calculs est projeté en termes de composants Wcomp dans la plateforme (cf. 3.2).

3.1. Langage d'aspects

La définition des modèles d'assemblage repose sur des ensembles de règles. Une règle porte sur des points de jonctions : « émission d'évènements » ou « réception de message ». Le langage d'expression des contrôles (*advices*) doit permettre la composition des règles et le contrôle des assemblages obtenus.

```

interaction switch_light_on(Switch switch, Light light) {
    switch.^on() :-
        switch._call() // light.on()
}

interaction switch_light_off(Switch switch, Light light) {
    switch.^off() :-
        switch._call() // light.off()
}

interaction alarm(Detector detector, Alarm alarm) {
    detector.^somebody() :-
        detector._call() // alarm.on()
}

interaction light_and_detector(Detector detector, Light light) {
    Boolean present := true;
    detector.^nobody() :-
        detector._call() // present := false // light.off()
    detector.^somebody() :-
        detector._call() // present := true
    light.on() :-
        if ( present ) light._call()
}

interaction light_control(Sensor sensor, Light light, int val, Shutter shutter) {
    Integer intensity := 0;
    sensor.^value(integer v) :-
        value._call( v ); intensity := v ;
    light.on() :-
        if ( intensity > val )
            shutter.open()
        else
            light._call()
}

interaction help_user(Sensor sensor, Light light, int valMin, Shutter shutter) {
    sensor.^value(integer intensity) :-
        value._call( intensity )
    if ( intensity < valMin )
        light.on() // shutter.close()
}

interaction delay(Sensor sensor, Timer timer, List events) {
    Boolean started := false
    timer.^started() :- timer._call() ; started := true
    timer.^stopped() :- timer._call() ; started := false
    timer.^handler() :- sensor._call()
    sensor.^^( ) :- if ( not started ) timer.start()
}

```

Figure 9 : Expression d'assemblage et langage d'aspects dédié.

Contrairement à différents travaux relatifs aux aspects et aux composants [8][18], nous nous situons dans un cadre réduit de la programmation par aspects. Notre objectif n'est pas de contrôler les méta-comportements tels que les connexions/déconnexions de composants ou l'ajout d'opérations, mais d'utiliser ces supports pour implémenter des points de jonctions plus proche de l'utilisateur. Ainsi, la définition d'un modèle d'assemblage correspond à la définition d'un aspect [7]. Elle prend en charge une préoccupation de l'application qui ne peut pas être anticipée. Elle contrôle simultanément plusieurs composants. L'expression des assemblages de composants repose sur la désignation des composants. Le tissage des aspects liés à l'assemblage lui-même est réalisé indépendamment de la plateforme cible. Les calculs de validité des assemblages sont alors opérés au niveau de la représentation que l'on construit de l'assemblage.

Nous avons choisi d'exprimer les aspects sous la forme de règles ISL que nous avons étendues pour permettre le contrôle des évènements [1]. Ainsi les aspects décrits dans la section précédente s'expriment par les codes suivants.

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

Le tissage des aspects est pris en charge par la plate-forme *Noah* en se basant sur les travaux formels de composition des règles ISL qui utilisent le `_call` comme un

pivot pour le calcul de composition d'aspects portant sur la réception d'un même message. . Si nous reprenons la configuration de la Figure 3, le résultat de la fusion des deux aspects a1' qui sont appliqués au même composant interrupteur donne lieu à une règle instanciée de la forme :

```

{
    i1.^off() :-
        i1._call() // light1.off() // light2.off()
}

```

Les deux lampes constituent implicitement un ensemble de composants associés [22]. Par contre, dans la configuration présentée par la Figure 4, la fusion des schémas n°5 sur l'allumage de la lampe détecte au point de vue des règles de fusion un conflit. En effet, l'ordre d'allumage de la lampe est considéré comme délégué à la fois à l'ouverture du premier et du deuxième volet. Dans ce cas, nous ne savons pas a priori composer les deux délégations. La composition dépend des choix de l'utilisateur. Il peut choisir entre ouvrir un ou les deux volets. Dans la description donnée au paragraphe 2.1, la composition attendue était une demande en parallèle d'ouverture de chacun des volets. Nous enregistrons la demande de l'utilisateur que nous mémorisons comme un assemblage de délégations.

3.2. Projection des aspects

La définition des stratégies sous forme de règles ISL est un moyen de décrire au niveau d'un modèle les assemblages de composants. Mais notre objectif n'est pas d'embarquer dans l'exécution les évaluations telles que celles de la plate-forme Noah [1] afin de minimiser le coût de l'interprétation à l'exécution. Nous avons donc fait le choix de transformer les règles ISL en des connexions entre composants Wcomp. Certains composants sont générés pour prendre en charge les différents cas exposés ci-dessus (contrôleurs, capteurs passifs, timers).

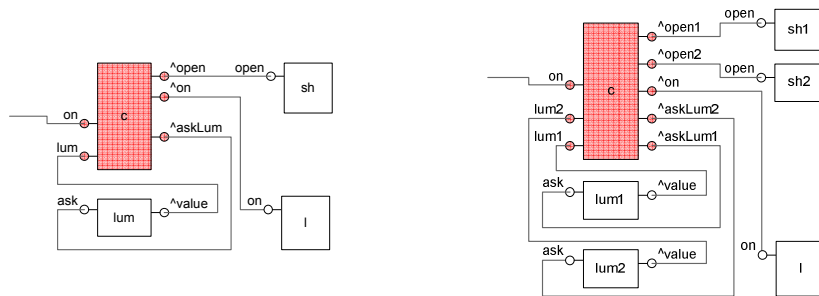


Figure 10 : Allumage ou ouverture d'un volet à gauche, de plusieurs à droite

La composition des aspects a1' présentée ci-dessus nous donne l'architecture décrite dans la **Figure 5** qui n'introduit pas de nouveaux composants alors que lorsque plusieurs aspects ont été composés sur un même point de jonction, il est nécessaire de générer des composants intermédiaires, dits contrôleurs. Nous avons fait le choix de générer des composants « advice », résultats du tissage uniquement lorsque c'est nécessaire.

Exemple : *Moduler l'ouverture d'une fenêtre ou l'allumage d'une lampe en fonction de la présence*

Le code relatif au composant contrôleur c est généré en parcourant l'arbre ISL correspondant au résultat de la fusion des règles définies par les aspects. Il sert d'intermédiaire à la lampe l et au volet sh .

Génération des propagateurs dédiés. Lorsque plusieurs composants matériels (modélisés du point de vue logiciel par des instances ayant les mêmes interfaces client et serveur) doivent être adressés, il est nécessaire de différencier l'adressage de ces composants. Pour ce faire, une technique consiste à générer de nouveaux propagateurs d'événements. Ces propagateurs sont générés à l'intérieur des composants.

Exemple : *Moduler l'ouverture de deux volets ou l'allumage d'une lampe en fonction de la présence*

Le problème visé ici est le suivant : en fait, il s'agit de découpler les événements de commande des volets ($sh1$ et $sh2$). C'est pourquoi, deux événements supplémentaires et différents sont créés ($\wedge open1$ et $\wedge open2$). Le contrôleur c est généré automatiquement par les règles ISL.

3.3. Evolution dynamique des assemblages

Dans l'application ciblée, les aspects sont appliqués en fonction du contexte d'usage. Il ne s'agit pas dans cet article de discuter des stratégies mises en place dans ce cadre, mais d'explicitier le processus d'évolution des assemblages. L'application d'un aspect consiste à calculer au niveau de la représentation de l'assemblage, le tissage des contrôles. Ce travail est pris en charge par le serveur d'interactions Noah qui a été étendu pour différencier les deux sortes de points de jonctions (émission d'événements et réception de messages). Au niveau de la projection, il s'agit de générer des nouveaux contrôleurs, et/ou de remplacer les anciens contrôleurs et/ou de modifier les liaisons entre les composants. Lors de cette dernière étape, l'ensemble de l'application embarquée est arrêté le temps des reconnections.

Le retrait dynamique d'un composant ou de l'application d'un aspect implique à nouveau un calcul des contrôles. En effet, les aspects qui sont appliqués à ce composant enlevé doivent être enlevés à leur tour. Le tissage est recalculé et les conséquences au niveau de la projection sont équivalentes à l'ajout défini précédemment.

3.4. Retour sur la solution proposée

Il n'a pas été nécessaire de modifier le modèle de composants Wcomp pour intégrer les aspects contrairement à l'approche de FRACTAL-AOP [3]. De plus, la lecture de l'assemblage de composants obtenu nous semble plus facile dans le modèle Wcomp. Le résultat de l'application des aspects engendre moins de composants que de règles. Les aspects ne sont pas réifiés au niveau de la plateforme, mais restent manipulables au niveau de la représentation.

Le contrôle à l'exécution des assemblages de composants obtenus correspond à une adaptabilité dynamique [3][17]. A chaque règle correspond au plus un composant ajouté. Nous opérons donc la projection uniquement par transformation de programmes. Mais nous gardons un lien entre la projection et la représentation des aspects. Ainsi l'évolution dynamique des assemblages est calculée au niveau du langage et projetée au niveau de la plate-forme Wcomp. Ces transformations ne sont pour l'instant pas formalisées mais seulement programmées. Notre approche se situe ainsi à la jointure entre la programmation par aspects et la programmation dirigée par les modèles.

4. Conclusion et perspectives

Nous avons donc exprimé l'évolution de ces applications à un niveau déclaratif avec notre langage d'aspects d'assemblage. Leur composition dans la plate-forme Wcomp a permis de répondre à certains besoins des systèmes embarqués qui sont : l'économie de la mémoire en évitant l'instanciation de composants inutiles et leur évolution logicielle dynamique.

Ensuite, les aspects sont le plus souvent utilisés pour capturer les propriétés dites non fonctionnelles des programmes. Nous avons montré dans cet article que la composition des composants pour répondre à des objectifs fonctionnels nécessitait également une expression séparée des préoccupations et un tissage des aspects. L'usage d'ADL pour décrire les architectures logicielles est aujourd'hui largement répandu. Néanmoins, il ne permet pas une expression séparée des assemblages. Une projection possible du langage d'aspect brièvement décrit dans cet article pourrait être une description avec un ADL et des composants générés. Une approche alternative consisterait à décrire des coupes par composants d'aspect [18].

Nous avons choisi de définir les aspects en utilisant un langage dédié indépendant, pour la partie « advice », de la plate-forme cible. Nous défendons l'idée que dans la mouvance de l'ingénierie dirigée par les modèles, les aspects doivent rester indépendants des langages des cibles. Ce choix permet d'analyser les parties *advice* des aspects. A notre connaissance, l'expressivité offerte par les langages de programmation est peu utilisée aujourd'hui dans ces parties. En l'occurrence dans le contexte de notre application, les calculs de tissages opérés au niveau d'une représentation de l'application nous permettent de détecter des conflits. En cela, nous nous rapprochons de différents travaux visant à intégrer une approche du développement logiciel basée sur la séparation des préoccupations [23].

Notons que lorsque certains aspects présentent des informations implicites nous pouvons avoir des problèmes³. Il nous reste à étudier plus de cas, pour établir ou non

3. Par exemple si au lieu d'explicitement l'aspect a4 comme un contrôle sur la lampe, nous l'associons directement à l'interrupteur : « à l'émission de l'évènement *SwitchOn*, s'il y a suffisamment de lumière, ouvrir les volets sinon allumer la lampe ». Il n'est pas dit de ne pas allumer la lampe. En conséquence, l'application simultanée de a1 et a4 modifié occasionne

la pertinence de telles propriétés par analyse des aspects. De même, une représentation des aspects au niveau modèle permet d'identifier des cycles potentiels. Ces cycles peuvent être conditionnés par les valeurs des capteurs. Ainsi dans notre exemple de mauvais choix de valeurs peuvent conduire à l'ouverture et la fermeture successives des volets tant que la valeur de la luminosité n'est pas soit suffisante pour garder les volets ouverts soit insuffisantes pour les garder fermés. Une des perspectives du travail décrit dans cet article est une détection des cycles.

Bibliographie

- [1] L. Berger. *Mise en œuvre des interactions en environnements distribués, compilés et fortement typés: le modèle MICADO*. PhD thesis, Université de Nice-Sophia Antipolis, octobre 2001.
- [2] M. Blay-Fornarino, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Un service d'interactions : principes et implémentation. *RSTI - série TSI*, 23(2):175-204, 2004.
- [3] Noury M. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. *Technique et Science Informatique*, 20(4), 2001.
- [4] Olivier Charra. *Conception de noyaux de systèmes embarqués reconfigurables*. PhD thesis, Université Joseph-Fourier, 2004.
- [5] Djalel Chefrour and Françoise André. Auto-adaptation de composants ACEEL coopérants. In *3ème Conférence Française sur les Systèmes d'Exploitation (CFSE'3)*, 2003.
- [6] D. Cheung, J. Fuchet, F. Grillon, G. Joulie, and J.-Y. Tigli. Wcomp : Rapid Application Development Toolkit for Wearable computer based on Java. In *IEE International Conference on Systems, Man and Cybernetics*, 2003.
- [7] Rémi Douence. A restricted definition of AOP. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004
- [8] H. Fakih and N. Bouraqadi. Les aspects et les composants logiciels - étude de cas avec le modèle de composants fractal. In *First French Workshop on Aspect-Oriented Software Development(JFDLPA 2004)*, Paris, France, September 2004.
- [9] D. Gay, P. Levis, R.v. Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language : A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation PLDI'03*, 2003.
- [10] G. Hjalmytsson and R. Gray. Dynamic C++ Classes - A lightweight mechanism to update code in a running program. In *Proceedings of the 1998 USENIX Technical Conference*, 1998.

l'allumage de la lampe et l'ouverture des volets. Cela nous conduit à envisager des interrelations entre les aspects. Dans cet exemple, a4 est absorbant vis-à-vis de a1.

- [11] John Keeney and Vinny Cahill. Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework. In *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks 'POLICY 2003'*, 2003.
- [12] Abdelmadjid Ketfi and Noureddine Belkhatir. Dynamic Interface Adaptability in Service Oriented Software. In *WCOP 2003*, 2003.
- [13] Abdelmadjid Ketfi and Noureddine Belkhatir and Pierre-Yves Cunin. Dynamic updating of component based applications. In *SERP'02*, 2002.
- [14] Abdelmadjid Ketfi. *Une Approche Générique pour la Reconfiguration Dynamique des Applications à base de Composants Logiciels*. PhD thesis, Université Joseph-Fourier, 2004.
- [15] Stéphane Lavirotte, Diane Lingrand et Jean-Yves Tigli. Définition du contexte : Fonctions de coût et méthodes de sélection, *UbiMob 2005*, Grenoble France, Juin 2005.
- [16] Scott Malabarba and Raju Pandey and Jeff Gragg and Earl Bar and J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *ECOOP 2000*, 2000.
- [17] Renaud Pawlak and Lionel Seinturier, Laurence Duchien, and G. Florin. Jac: A flexible and efficient solution for aspect-oriented programming in java. In *Reflection 2001*, volume 2194 of *Lecture Notes in Computer Sciences*, page 1–24, Kyoto, Japan, September 2001. Springer Verlag.
- [18] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Olivier Barais. Une extension de fractal pour l'aop. In *Première journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2004)*, Paris, France, September 2004.
- [19] Pierre-Guillaume Raverdy and Hubert Le Van Gong and Rodger Lea. DART: A Reflective Middleware for Adaptive Applications. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, 1998.
- [20] Barry Redmond and Vinny Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. In *ECOOP'2000 Workshop on Reflection and Metalevel Architectures*, 2000.
- [21] Javier Resano and Daniel Mozos. Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, 2004.
- [22] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In Karl Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 16-25. ACM Press, March 2004.
- [23] Francis Tessier, Mourad Badri, and Linda Badri. Détection de conflits entre aspects - de la modélisation à l'analyse du code. In *Première journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2004)*, Paris, France, September 2004.