CHAPITRE 3

WCOMP

Coordonnateur : Jean-Yves Tigli.
Rédacteurs : Vincent Hourdin, Stéphane Lavirotte.

Ce chapitre présente les modalités de prise en charge des différentes contraintes dans la plate-forme WComp.

3.1 Introduction

Le modèle WComp, dont nous présentons l'implémentation C♯ .NET, est une architecture à composants légers basée sur des communications par événements. Deux types d'entités sont définies dans notre approche :
– des containers permettant d'exécuter une application à base de composants en les instanciant et créant des liaisons entre eux,
– des designers permettant de modifier les données des containers en modifiant les assemblages présents dans les containers.
De même que pour la communication entre les composants, les événements ont une place majeure dans le modèle : les designers sont notifiés des changements d'états des containers pour pouvoir prendre des décisions quant aux adaptations à effectuer, et ils ordonnent les modifications architecturales des applications en envoyant des événements au(x) container(s).

Les interfaces externes des containers permettent de les connecter aux designers, mais aussi à d'autres containers. On peut d'ailleurs souligner que le designer sur lequel nous allons baser nos exemples est en fait un assemblage de composants, et donc s'exécute dans un container. Il apparaît alors possible de contrôler un container, son architecture et l'application qu'il exécute, à partir d'un autre, ce qui fait naître une hiérarchie dans le modèle.

Les exemples de code qui seront décrits pour les contrats correspondent au code utilisé par un designer réactif de la plate-forme : le designer d'aspects d'assemblage. En se basant sur les événements envoyés par les containers et un ensemble de règles (les aspects d'assemblages), il va modifier automatiquement et de façon déterministe l'assemblage de composants du container auquel il est relié. La syntaxe des règles est proche d'une syntaxe d'aspects, elle définit des points de coupe sur les composants comme étant les ports d'entrée et de sortie (respectivement méthodes et événements). Le greffon est, quant à lui, un sous assemblage qui va venir s'ajouter ou se substituer à l'assemblage qui s'exécute dans le container. Ce sont ces règles, appelées aspects d'assemblages, que nous allons écrire pour prendre en compte les contrats décrits dans le chapitre 1.3.

Il est utile de préciser que la plate-forme WComp est constamment en exécution, et totalement dynamique. L'ajout d'un contrat est donc une incrémentation et non un redéploiement de l'application (ADL) ou de la plate-forme.
Enfin, la portée du modèle WComp se résume aux containers et aux designers interagissant pour créer l'application. Si l'application utilise des dispositifs ou des services de l'environ-

nement, ils sont subits par l'application, et leur fonctionnement ne peut être déterminé par celle-ci. La portée du modèle se limite aux composants proxy de ces services distants. La limite de granularité est donc le composant, le modèle adoptant la notion de composant "boite noire" qui fournissent des fonctionnalités par une interface, mais dont on ne connait pas l'implémentation et dont on ne peut modifier le code.

3.2 Prise en charge des évènements

Comme nous l'avons vu, toutes les communications sont réalisées sous forme d'événements, et effectuer des opérations sur certains événements peut se traduire par une modification de l'architecture de l'application, en ajoutant des composants, des liaisons, ou encore des containers et designers.

Le modèle WComp prend complètement en charge 7 des 9 événements décrits dans le chapitre 1.2, et les deux autres de façon limitée.

OperationEntryEvent et OperationExitEvent :

Ces événements ne peuvent pas directement exister dans cette plate-forme, puisqu'ils nécessiteraient de se placer dans le code des composants, et du fait l'adoption de composants boites noires (portée considérée), fournir ces événements est impossible. En revanche, il est possible d'intercepter une invocation avant l'arrivée de celle ci, ainsi que son retour, ce qui correspond aux événements RequestSending et RequestReceiving du paragraphe suivant. Cependant, les liaisons étant implémentées comme une invocation simple de méthode, il n'y a qu'une infime différence entre l'envoi d'une requête et l'entrée dans le composant ciblé.

Bien qu'au niveau d'un composant, WComp ne puisse pas modéliser ces événements, si on se place dans un modèle hiérarchique où l'on considère un container comme un composant, il est possible d'avoir ce genre de notifications. Cela prend d'ailleurs tout son sens, puisqu'il y a une couche de communication entre l'envoi de la requête et sa réception (l'entrée dans l'opération). Donc WComp prend en charge ces deux événements, de façon limitée, suivant le niveau où l'on se place.

RequestSending et RequestReceiving :

Il est possible de créer ces deux événements en interceptant une invocation, c'est à dire en modifiant l'assemblage de composants pour introduire de nouveaux composants sur la liaison. L'ordre d'exécution étant pris en compte et déterministe dans la plate-forme, il est aussi possible d'obtenir un événement RequestReceiving caractérisant la fin d'une exécution.

Comme nous l'avons vu, les designers commandent les containers pour instancier des composants ou créer des liaisons et les supprimer par exemple. Il est donc possible ici aussi d'intercepter l'invocation des commandes des containers en positionnant un composant de précondition sur le flot de contrôle. C'est la manière que nous utiliserons pour les événements BeforeStartExecution, BeforeBindingAdding, BeforeRetractService et BeforeRegisterService.

BeforeStartExecution et BeforeRegisterService :

Ces événements sont identiques pour notre plate-forme, ils correspondent tous les deux à l'interception de l'instanciation d'un composant.

BeforeBindingAdding :

Cet événement correspond à l'interception de l'invocation d'un ajout de composant.

BeforeRetractService : on procède de la même manière avec la méthode de suppression de composant.

ApplicationEvent :

La réaction sur les événements internes à un assemblage de composants est rendu possible par l'utilisation de composants sondes. Ces derniers permettent d'exporter de nouvelles fonctionnalités dans l'interface externe du container dans lequel ils sont instanciés, en entrée (puits) ou en sortie (source). Il est donc possible de réagir à ces événements par recomposition de l'application ou toute autre action existant dans la plateforme.

3.3 Contraintes

3.3.1 Respect d'un délai donné

Code

Il est par exemple possible de vérifier le temps d'exécution à un web service, en appliquant le contrat au composant représentant le proxy du web service. L'exemple suivant envoie un événement d'erreur indiquant que le contrat n'est pas respecté si l'invocation à un service météo prend plus de temps que le paramétrage du composant de timeout.

Version1 Dans cette version, on place le contrat de respect de délai dans un container, et l'invocation du service météo et le traitement des données dans un autre. Ce dernier sera vu comme un proxy pour web service pour dispositif dans le premier. C'est nécessaire pour pouvoir obtenir les événements de type OperationEntryEvent et OperationExitEvent. De plus, cela permet d'effectuer directement des changement architecturaux autour de l'invocation du service s'il prend trop de temps à s'exécuter. Dans l'exemple suivant, l'aspect d'assemblage ayant pour nom 'Slow' sera activé par la notification d'échec du contrat ; il aura par exemple pour but de ne plus continuer son invocation le temps que sa charge diminue, ou spécifier un autre service pour obtenir les informations de météorologie.

```
schema ContratTimeout1(wcompApplication,wcompAADesigner,contratTimeout,textBox):

  wcompApplication.^GetMeteoBegin -> (
    contratTimeout.Start
  )

  wcompApplication.^GetMeteoEnd -> (
    contratTimeout.Check
  )

  contratTimeout.^Error -> (
    wcompAADesigner.EnableSchema("Slow") ; textBox.set_Text
  )
```

Version2 On vérifie ici simplement le temps d'exécution du service météo en mesurant le temps à partir des liaisons de l'assemblage.

```
schema ContratTimeout2(meteoProvider,contratTimeout,textBox):

  meteoProvider.GetMeteo -> (
    (contratTimeout.Start; call)
  )
```

```
  meteoProvider.^GetMeteoReturn -> (
    if (contratTimeout.Check)
      { call }
    else
      { nop }
  )

  contratTimeout.^Error -> (
    textBox.set_Text
  )
```

3.3.2 Authentification

Pour l'authentification, nous considérons que pour chaque invocation de méthode devant être authentifiée, le premier argument contient les informations nécessaires à l'authentification. Il suffit alors d'effectuer un test auprès du service d'authentification avant d'effectuer réellement l'invocation authentifiée. Nous n'utilisons pas le composant classique du IF des aspects d'assemblage, car il faut faire transiter le premier argument de l'appel intercepté dans la demande d'authentification et effectuer l'invocation avec un argument de moins ; nous utilisons un composant cond qui a cet effet.

Code

```
schema ContratAuth(meteoProvider,authentifier,textBox):

  meteoProvider.GetMeteo -> (
    cond (authentifier)
      { call }
    else
      { nop }
  )

  authentifier.^Error -> (
    textBox.set_Text
  )
```

3.3.3 Capacité des devices et taille des données

Ce contrat est assez proche du précédent, puisqu'il nécessite la vérification d'un critère avant d'autoriser la réelle invocation d'une méthode. Nous supposons qu'un service ou un composant nous donne les capacités des dispositifs et nous indique si une valeur correspond à ce qu'un dispositif peut supporter.

Code

```
schema ContratCapacity(meteoProvider,capacity,textBox):

  meteoProvider.GetMeteo -> (
    cond (capacity)
      { call }
    else
      { nop }
  )
```

```
  capacity.^Error -> (
    textBox.set_Text
  )
```

3.3.4 Appel à un service ou composant exigée

La plate-forme WComp ne permet de vérifier ce type de contrat que de façon limitée. L'ajout de trace d'exécution dans un assemblage de composants est possible, en ajoutant un composant de trace pre/post en séquence sur chaque liaison. La vérification d'un appel à un composant exigé nécessiterait alors d'étudier ces traces. La mise en place d'une telle infrastructure de trace est assez lourd, car les aspects d'assemblage dépendent d'un composant, et on ne peut pas en appliquer un sur tous.
La deuxième façon de remplir ce contrat nécessite d'étudier le code des composants, ce qui n'est pas possible puisque le modèle WComp se base sur des composants boite noire. Cependant, il est possible de vérifier la présence d'une liaison entre deux composants, et donc la présence d'un appel à un composant exigé.

Code

Le code suivant est un exemple d'implémentation de la première solution. La deuxième est proche de l'implémentation du contrat 6, "Limitation sur les connexions entre composants ou services" dans lequel il est aussi nécessaire d'obtenir la liste des liaisons de l'assemblage de composants.
Le contrat vérifie que le service d'authentification a bien été appelé avant le service météo. Nous partons du principe que le composant de log est appelé avant et après chaque invocation sur un autre composant, en transmettant le nom de la méthode invoquée.

```
schema ContratCheckLog(meteoProvider,log,textBox):
  meteoProvider.GetMeteo -> (
    if (log.hasBeenCalled("auth")
      { call }
    else
      { nop }
  )

  log.^Error -> (
    textBox.set_Text
  )
```

3.3.5 Présence d'un service ou composant exigée

Ce contrat peut prendre diverses formes, puisque l'expression le caractérisant est variable. Avec WComp, il est possible de vérifier la présence d'un composant dans un assemblage par son nom (une instance), son type (la possibilité de l'instancier), ou encore de vérifier que son interface est compatible avec celle d'un autre composant.
L'exemple suivant vérifie la présence d'un composant lors de l'ajout d'un composant, puisque comme nous l'avons vu dans le chapitre précédent, le démarrage d'une application dans WComp correspond à l'instanciation d'un composant. La demande d'instanciation par un designer est interceptée avant que le container ne l'effectue, et vérifie si celui qui nous interresse est dans la liste des composants déjà instanciés.

Code

```
schema ContratPresence(wcompContainer,wcompDesigner,strCompare,valueEmitter,textBox):

  wcompDesigner.^CreateComponent -> (
    wcompContainer.GetInstances ;
    if (strCompare.HasComponent)
      { call }
    else
      { ValueEmitter.^StringEvent }
  )

  wcompContainer.^GetInstancesReturn -> (
    strCompare.SetStrings
  )

  valueEmitter.^StringEvent -> (
    textBox.set_Text
  )
```

3.3.6 Limitation sur les Connexions entre composants ou services

Les aspects d'assemblage permettent de donner une exclusivité à une liaison grâce à un mot clé, *only*. Cela peut se produire dans le cas où il y a un conflit, c'est à dire où plusieurs aspects d'assemblage sont appliqués en même temps sur un assemblage de composants avec des points de superposition. Cependant, cette solution ne permet pas simplement d'obtenir un événement notifiant que le contrat n'a pas été respecté et qu'il a fallu en priviligier une, sans modifier le designer d'aspects d'assemblage.

La solution que nous mettons en oeuvre est donc semblable à la précédente, et consiste à vérifier la présence d'une liaison dans l'assemblage qui pourrait avoir un port commun avec la nouvelle liaison que l'on souhaite instancier.

Code

```
schema
ContratLimitation(wcompContainer,wcompDesigner,linkHasSamePort,valueEmitter,textBox):

  wcompDesigner.^CreateLink -> (
    wcompContainer.GetLinks ;
    if (linkHasSamePort.HasIt)
      { call }
    else
      { valueEmitter.^StringEvent }
  )

  wcompContainer.^GetLinksReturn -> (
    linkHasSamePort.SetLinks
  )

  valueEmitter.^StringEvent -> (
    textBox.set_Text
  )
```

3.3.7 Contrôles du systèmes sur événements applicatifs

Puisque toutes les communications sont des événements dans WComp, avoir un contrat qui effectue un contrôle lors de l'émission d'un événement applicatif ne nécessite pas de modification de la plate-forme. Si la sonde permettant de vérifier la contraite se trouve dans le même assemblage que la source de l'événement applicatif, un simple sous-assemblage de composants permettra de créer la contrainte. Si ce n'est pas le cas, il est possible de remonter l'événement en dehors du container, en utilisant des composants dits sondes qui ajoutent des ports d'entrée/sortie à l'interface du container.

Code

L'exemple suivant entrainera une modification de l'application par application d'un aspect d'assemblage lorsque l'événement "batterie faible" sera émis par un composant présent dans le container, et relié à un composant sonde permettant de le faire apparaitre dans l'interface externe du container. Un container a en réalité deux interfaces, une de controle, que nous avons utilisé pour réaliser les contrats précédents, et une applicative, que nous utilisons ici.

```
schema ContratAppliEvent(wcompApplication,wcompAADesigner):
  wcompApplication.^LowBattery -> (
    wcompAADesigner.EnableSchema("LowBattery")
  )
```

3.3.8 Contrainte applicative composite

Dans WComp, une liaison ne peut exister si elle n'est pas reliée à deux ports : un événement et une méthode. Vérifier qu'un composant existe avant de vérifier la présence d'une liaison est donc inutile, puisque si la liaison existe, le composant existe forcément. Nous pouvons donc réaliser ce contrat de la même façon que celui d'appel à un composant exigé (3.3.4).
Cependant, la composition de contrats, puisqu'ils sont écrits sous forme d'aspects d'assemblage, peut être faite simplement par le moteur de composition et de fusion des aspects d'assemblage. Les aspects d'assemblage utilisent des opérateurs bien spécifiés, pour lesquels les relations d'associativité, de commutativité et d'idempotence sont vérifiées. La composition de ces aspects a donc un résultat déterministe.
Dans leur écriture, on retrouve les séquences ';', les ordres indéfinis '+' ou le parallélisme, l'appel original avec le mot clé 'call', ou encore l'absorption des autres règles par le mot clé 'only'. Des priorités et ordres d'applications peuvent être ainsi créés.