CHAPITRE 5

Plates-formes

Coordonnateur : Laurence Duchien.
Rédacteurs : Mireille Blay-Fornarino, Philippe Collet, Vincent Hourdin, Stéphane Lavirotte, Sébastien Mosser, Nicolas Rivierre, Lionel Seinturier, Jean-Yves Tigli .

Ce chapitre illustre l'utilisation des éléments précédemment définis sur chacune des plates-formes cibles du projet FAROS. Chaque plate-forme se trouve ainsi caractérisée par les événements du métamodèle pivot qu'elle est capable de prendre en compte, et par la façon dont elle met en œuvre la vérification de différentes formes de contrat. Par la suite, nous détaillons, pour chaque plate-forme, un exemple illustrant une transformation possible jusqu'au code opérationnel, et un autre exemple illustrant certaines difficultés ou impossibilités à effectuer une transformation.
Nous faisons d'abord un rappel des événements pris en charge par chacun des plates-formes, tel que cela a déjà été défini dans le livrable F-3.2.

	Adore-Coconet		Wcomp	ConFract	AOKell-FAC		Orqos
	Version 1.0	Version 2.0			Version 1.0	Version F4F	
OperationEntry	ok	ok	limité	ok	ok	ok	ok
OperationExit	ok	ok	limité	ok	ok	ok	ok
RequestSending	non	limité	ok	limité	ok	ok	ok
RequestReceiving	non	limité	ok	limité	ok	ok	ok
BeforeStartExecution	non	non	ok	ok	non	ok	limité
BeforeBindingAdding	limité	limité	ok	limité	non	ok	limité
BeforeRetractService	limité	limité	ok	limité	non	ok	limité
BeforeRegisterService	non	non	ok	limité	non	ok	limité
ApplicationEvent	limité	limité	ok	v2	ok	ok	limité

Nour reprenons la liste que nous avons proposée au chapitre 3 dans la table ci-dessus avec quelques événements supplémentaires et nous complétons cette même table 5.1 avec les informations de gestion de ces événements par les plates-formes.

5.1 Validation de l'approche proposée

A partir de l'ensemble des événements que nous avons décrits dans le chapitre 3, du langage et du modèle contractuel définis dans le chapitre 4, nous proposons ici de vérifier pour chaque plate-forme, la faisabilité de la mise en œuvre des premiers exemples décrits. Nous ne donnons pas explicitement de modèle d'exécution, celui-ci étant très lié à la plate-forme. Cependant nous proposons dans l'annexe A, deux exemples de modèle d'exécution, l'un reposant sur une communication en mode synchrone et l'autre sur un mode asynchrone.

5.2 Adore - Coconet

Dans le cadre de cette étape de transformation, nous faisons le choix de limiter la cible à la partie orchestration de la plate-forme Adore. La prise en compte des évènements est donc modifiée (cf. Tableau 5.1).
La partie de la plate-forme Adore que nous ciblons se présente comme un framework de définition d'architectures SOA à base de web services et d'orchestrations BPEL. Un concept

	Adore-Coconet	Wcomp	ConFract	AOKell-FAC		Orqos
				Version 1.0	Version F4F	
Bind	limité	ok	limité	non	ok	limité
Unbind	non	ok	non	non	ok	
AfterBind	non	ok	non	non	ok	
AfterUnbind	non	ok	non	non	ok	
RegisterEntity	non	ok	limité	non	ok	limité
AfterRegisterEntity	non	ok	non	non	ok	
UnregisterEntity	non	non	limité	non	ok	limité
AfterUnregisterEntity	non	non	non	non	ok	
StartSystem	limité	non	ok	non	ok	limité
StopSystem	non	non	non	non	ok	
StartEntity	non	ok	ok	non	ok	limité
AfterStartEntity	non	ok	non	non	ok	
StopEntity	non	ok	non	non	ok	
AfterStopEntity	non	ok	non	non	ok	
RequestSending	oui	ok	limité	ok	ok	ok
ResponseReceiving	oui	ok	limité	ok	ok	ok
OperationEntry	ok	limité	ok	ok	ok	ok
OperationExit	ok	limité	ok	ok	ok	ok
ApplicationEvent	limité	ok	v2	ok	ok	limité

TAB. 5.1 – Correspondance entre évènements et émetteurs(e) ou récepteurs (r) de ces évènements.

supplémentaire, les évolutions, permet de contrôler les orchestrations.
Dans le cadre de Faros, nous présentons ci-après les principes de l'architecture ciblée à partir d'un modèle pivot comprenant la structure de l'application et les contrats.

5.2.1 Métamodèle de la plate-forme Adore

A partir des entités au niveau du pivot, des services sont générés au niveau de la plate-forme. Nous distinguons :
- les services "boîtes noires" qui correspondent à des implémentations des services que nous ne contrôlons pas.
- les services "boîtes blanches" qui sont des services que nous contrôlons.

Les contrôles s'expriment sous la forme d'évolution qui sont des spécialisations des orchestrations, dans lesquels les points d'accroches (hook, successor et predecessor) sont spécifiés.
La figure 5.1 présente le métamodèle correspondant aux orchestrations et évolutions dans Adore.

5.2.2 Modèles dans Adore relativement au pivot

A partir d'un modèle défini au niveau du pivot, nous générons une architecture à base de services. Pour permettre un contrôle atomique des services, nous générons devant tous les services (que ce soit des services générés ou extérieurs), une orchestration de contrôle qui est composée initialement d'une séquence : receive, invoke puis reply ; nous nommons par la suite ces services "proxy" même si le métamodèle ne les distingue pas.
Les contrats sont définis sous la forme d'évolutions. Une entité dîte *ContractManagerService* est notifiée chaque fois qu'une violation est détectée. Elle a la charge de gérer l'erreur détectée. Le

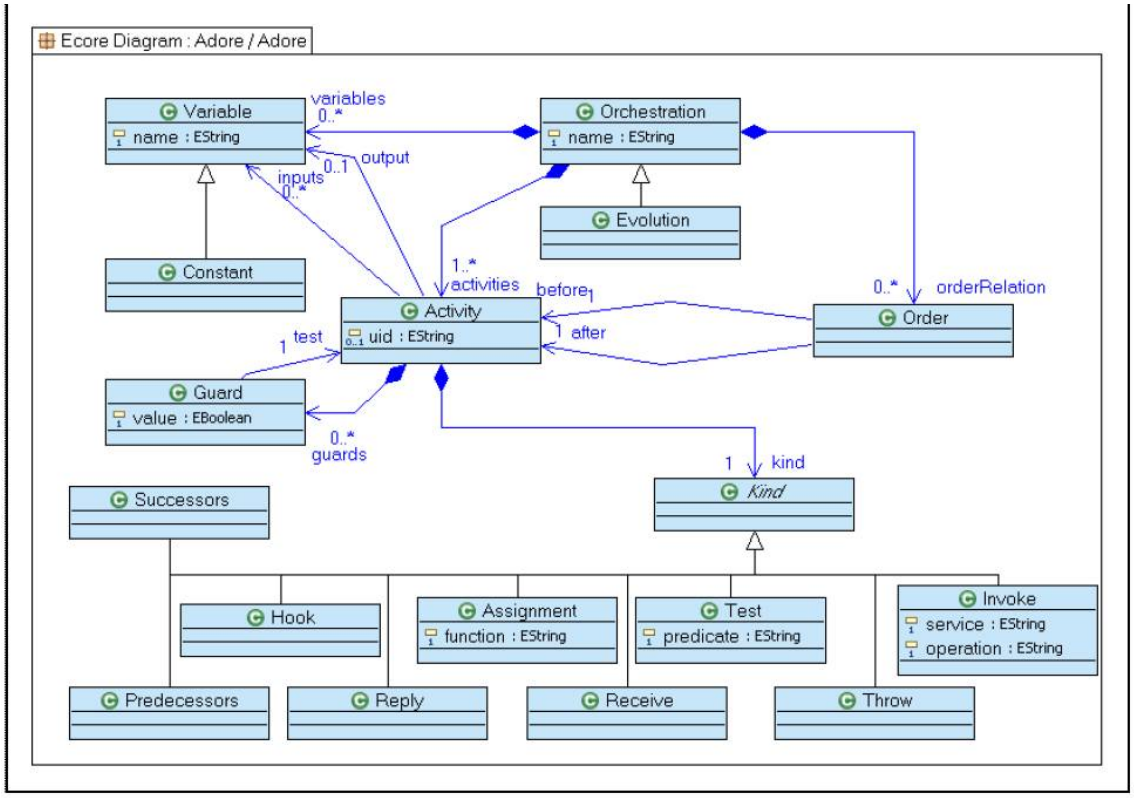


FIG. 5.1 – Métamodèle de la plate-forme Adore.

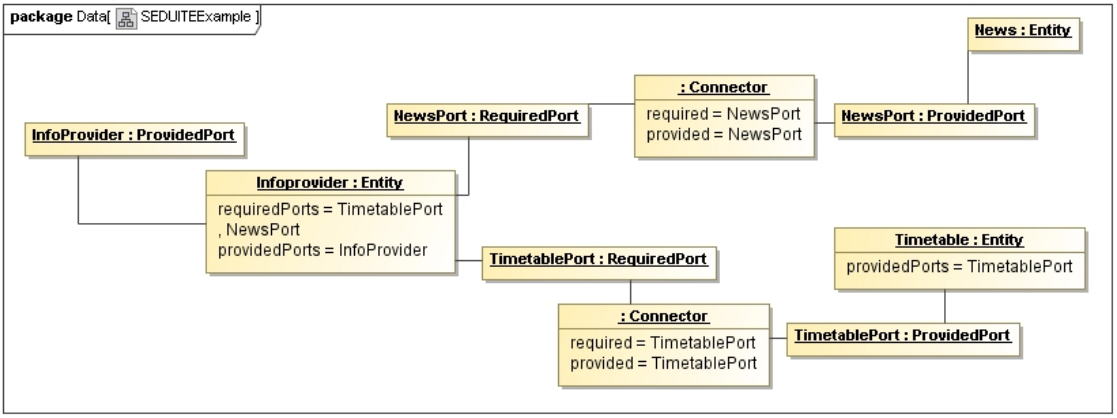


FIG. 5.2 – Exemple de modèle au niveau du pivot

positionnement de la vérification dans la définition du contrat au niveau du pivot détermine si l'activité contrôlée a lieu ou non.

Du pivot à la structure de l'application

A partir du modèle obtenu au niveau du pivot, nous procédons à un raffinement au cours duquel les appariements avec des services existants sont définis. Les orchestrations non encore présentes au niveau de la plate-forme sont alors générées conformément au métamodèle de Adore.

Exemple A partir du modèle défini au niveau du pivot (cf. Figure 5.2) et des informations supplémentaires suivantes : *Infoprovider*, nouveau service, invoque *news* qui est généré et *Timetable* qui est préexistant et concatène les résultats, nous aboutissons à 6 services dont 1 pré-existant (cf. Figure 5.3).
- *News* du pivot devient *RealNewsService* et *News*. *RealNewsService* reste à implémenter sur la base des interfaces générées, tandis que *News* est un service proxy offrant les mêmes interfaces que *RealNewsService* et dont chaque opération est définie par une orchestration élémentaire : `receive; invoke RealNewsService; reply.`
- *Timetable* devient *RealTimetableService* (au nommage près qui est celui du service pré-existant, lequel est déterminé par une phase de correspondance) et *Timetable*.
- *Infoprovider* devient *RealInfoProviderService* et *InfoProvider*. *RealInfoProviderService* est l'orchestration générée. Elle fait appel à *News* et *TimeTable*, et la concaténation est identifiée comme un *append*, pour le reste même procédé.

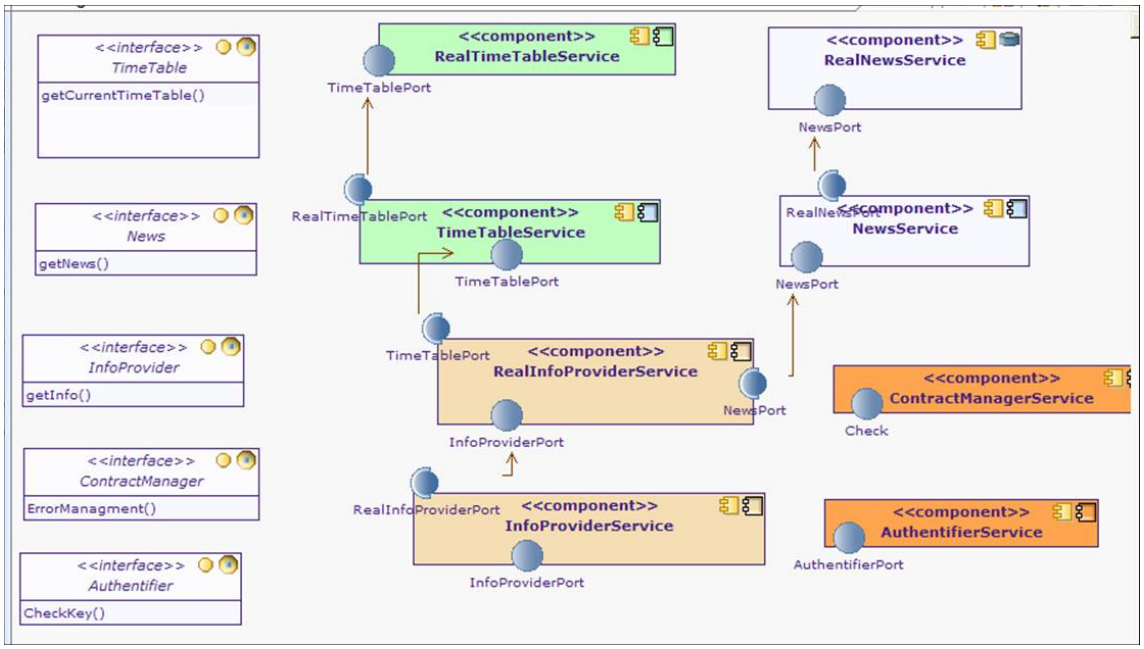


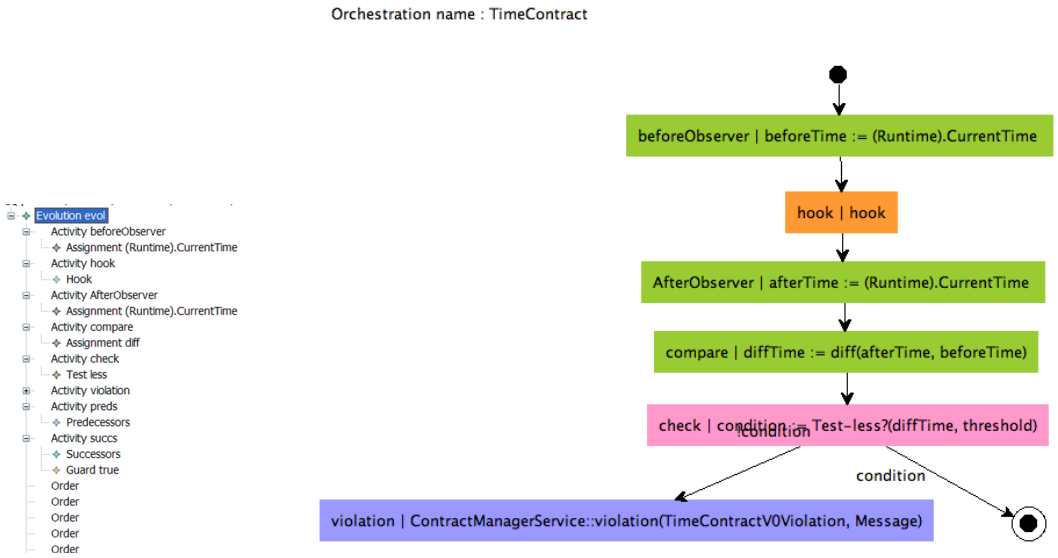
FIG. 5.3 – Modèle de l'exemple dans la plate-forme Adore.

Des contrats dans les orchestrations

Les contrats sont injectés dans la plate-forme sous la forme d'"évolutions". La démarche étant dirigée par les modèles nous passons directement par la génération de modèle ECORE conforme au métamodèle présenté par la figure 5.1. Pour faciliter la lecture, nous utilisons

l'outil DrAdore (cf. http ://rainbow.i3s.unice.fr/adore) pour visualiser les évolutions obtenues. Nous ne présentons donc pas le langage de surface, inutile dans cette étape vers Adore.

Contrats de temps La figure 5.4 visualise l'évolution et le binding correspondant au contrat *ContractTimer* de la version littérale. La même évolution est utilisée pour le contrat côté client mais cette fois le binding est réalisé sur toutes les invocations *InfoProviderService.infoProviderPort.getInfo*.



Cette évolution est appliquée par liaison sur l'invocation de *RealInfoProviderService.infoProviderPort.getInfo* pour le contrat 4.4.1 et par "binding" sur *InfoProviderService.infoProviderPort.getInfo* pour le contrat 4.4.4.

FIG. 5.4 – Evolution représentant le contrat *TimeContract* dans la plate-forme Adore.

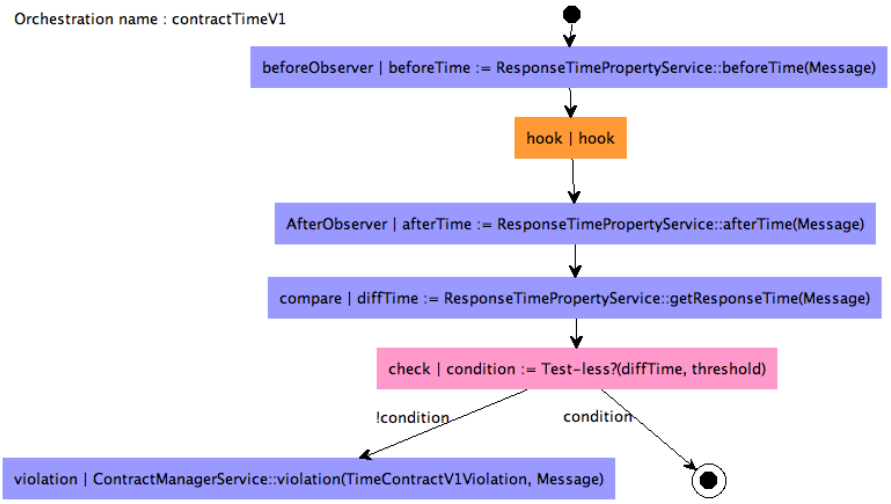
La figure 5.5 visualise l'évolution et le binding correspondant au contrat *ContractTimer* de la version avec propriété. La propriété est réifiée sous la forme d'un service.

La version par "checker" est une approche pré-implémentée de la première version ; en reconnaissant le checker, la transformation génère la première solution.

Contrat d'Authentification Le contrat d'Authentification (cf. 4.5) ne peut pas être pris en compte dans Adore s'il est écrit en utilisant la détermination dynamique du service (version 2), point que à ce jour nous ne prenons pas en compte. De plus la représentation de la propriété clef comme un service ne nous semble pas adéquate dans cet exemple. Cela nous conduit à envisager une transformation des propriétés associées à une opération et accédées uniquement en lecture comme pouvant également correspondre à un paramètre supplémentaire de l'opération (cf. Figure 5.6.

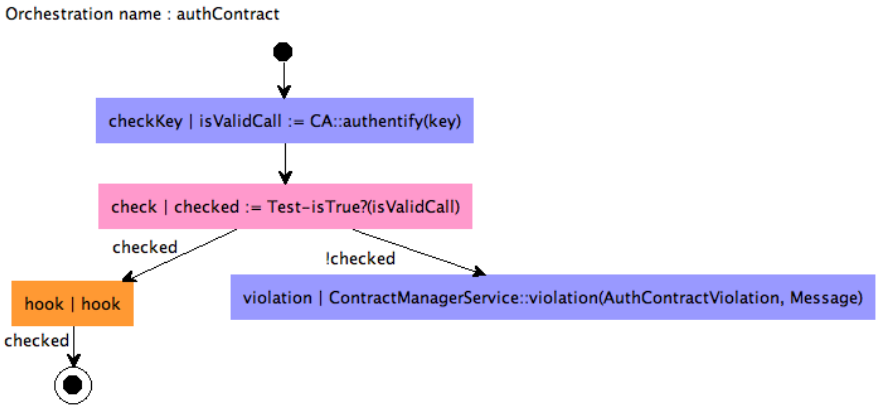
5.2.3 Généralisation et limitations

En conséquence, les événements de contrôle nous conduisent à positionner les points d'accroches des évolutions selon le tableau suivant.



Cette évolution est appliquée par liaison sur l'invocation de *infoProviderService.infoProviderPort.getInfo*

FIG. 5.5 – Evolution représentant le contrat *TimeContract* avec propriété (cf. 4.4) dans la plate-forme Adore.



Cette évolution est appliquée par "binding" sur l'invocation de *RealInfoProviderService.infoProviderPort.getInfo*)

FIG. 5.6 – Evolution représentant le contrat Authentification (cf. 4.5) dans la plate-forme Adore.

When Event On Element [do|Check] exp	Hook
OperationEntry on Entity Operation do exp	Proxy.invoke(RealEntity.Operation)
OperationEntry on Entity Operation check exp	Proxy.invoke(RealEntity.Operation)
OperationExit on Entity Operation do exp	Proxy.invoke(RealEntity.Operation)
OperationExit on Entity Operation check exp	Proxy.invoke(RealEntity.Operation)
RequestSending on Entity Operation do exp	dans toutes les orchestrations invoke(Entity.Operation)
ResponseReceiving on Entity Operation do exp	dans toutes les orchestrations invoke(Entity.Operation)

Les transformations relatives au Checker, boîte noire, sont définies au coup par coup. En particulier *BBProtocolOutChecker* devrait pouvoir être pris en compte statiquement par analyse du code des orchestrations. De même les contrôles relatifs à *startEntity* ou *Bind* devraient être vérifiés statiquement, ce qui est une limitation justifiée au regard du contrat précédemment défini puisque l'opération dynamique de binding n'existe pas dans notre plate-forme.

Limitations Nous ne prenons pas en compte l'invocation dynamique des services, ni l'ajout dynamique de service.
Lorsque nous sommes obligés de répartir un contrat sur plusieurs évolutions (par exemple parce qu'une observation porte sur *RequestSending*) et une autre sur *OperationEntry*) , nous ne garantissons pas l'atomicité de la pose des évolutions. En particulier des erreurs de fusions peuvent conduire à un état instable des contrats.

5.3 Wcomp

L'application du contrat de respect d'un délai donné sur les événements OperationEntry et OperationExit nous oblige à appliquer ce contrat sur un container, puisqu'il n'est pas compatible avec notre approche boîte noire du plus bas niveau. Plus clairement, l'opération sur laquelle il sera appliqué sera un service composite WComp. Pour les trois premières versions du contrat (coté serveur donc), nous allons modifier l'assemblage de composants du service composite, qui est une boîte blanche dans laquelle nous pouvons effectuer des mofications structurelles.

5.3.1 Version littérale sans propriété

La figure 5.7 illustre les modifications que nous allons effectuer par aspect d'assemblage dans le service composite d'origine. Les composants que nous ajoutons sont en hachuré. Les deux composants sur lesquels ont voit un ' ;' sont des composants de séquence, celui sur lequel est écrit 'CT' est le composant CheckTimer qui lance sauvegarde les dates d'entrée et de sortie dans le container et fait la différence à la fin et envoie un événement. Ce dernier est exporté vers le service composite par le composant marqué 'O' qui est un composant sonde source.
Voici le code de l'aspect d'assemblage qui permet ces modifications et la mise en oeuvre de ce contrat sur la plate-forme :
Pointcut :

```
sondeEntree:=/GetInfoProbe/
sondeSortie:=/GetInfoReturnProbe/
```

Advice :

```
schema ContratTimeout1(sondeEntree,sondeSortie):
  contratTimeout : 'Wcomp.Beans.ContratTimeout' ;
  sondeContrat : 'Wcomp.Probes.OutputProbe' ;

  sondeEntree.^Out -> (
```

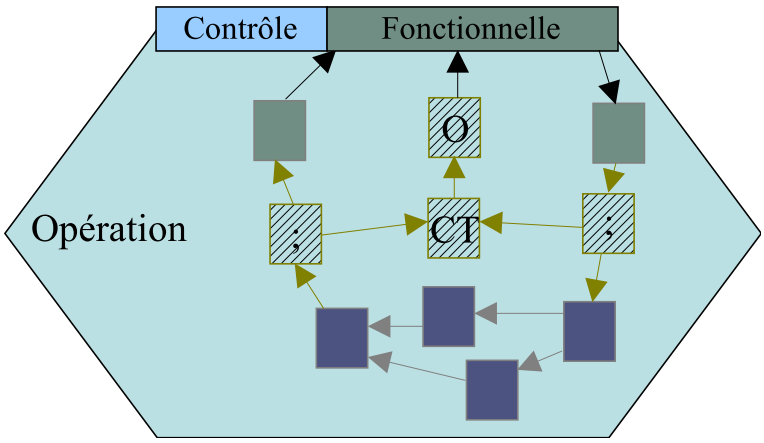


FIG. 5.7 – Illustration de la prise en charge du contrat Timer 1 dans WComp.

```
    contratTimeout.Start ; call
  )

  sondeSortie.In -> (
    contratTimeout.Stop ; call
  )

  contratTimeout.^Error -> (
    sondeContrat.In
  )
```

Le code du composant ContratTimeout ci-dessous est l'ajout de code à faire pour l'implémentation du contrat en lui même sur la plate-forme :

```
using System;
using WComp.Beans;
using WComp.EventedBeans;
using System.Threading;
using System.Collections;

namespace WComp.Beans
{
  [Bean(Category="Contract"]
  public class ContratTimeout : EventedDrawable {

    class MyThreadContext {
      private int id, timeout;
      public int counter;
      public string eid;
      public ContratTimeout ct;
      public DateTime startTime;
      private volatile int done = 0;

      public MyThreadContext(int _id, string _eid, DateTime _startTime, ContratTimeout
        id = _id;
        startTime = _startTime;
        ct = _ct;
        counter = _c;
        timeout = _timeout;
```

```
      this.eid = _eid;
    }

    public void reset() {
      done = 1;
    }

    public void ThreadLoop() {
      Thread.Sleep(timeout);
      if (done == 0) {
        ct.SError("Contract is not respected.");
      } else {
        ct.SError("Contract is respected.");
      }
    }
  }

  private int counter = 0, timeout = 5000;
  Hashtable table = new Hashtable();
  string last_start_event_id;

  public int Timeout_ms {
    get { return timeout; }
    set { timeout = value; }
  }

  public string getKeyIdFromParams(object[] o, int counter) {
    string result = "" + counter;
    if (o == null) return "null";
    foreach (object i in o) {
      result += i.GetType().Name + i;
    }
    return result;
  }

  public void Start(params object[] o) {
    counter++;
    last_start_event_id = getKeyIdFromParams(o,counter);
    int id = Thread.CurrentThread.GetHashCode();
    DateTime startTime = DateTime.Now;

    MyThreadContext tc = new MyThreadContext(id, last_start_event_id, startTime, this
    Thread t = new Thread(new ThreadStart(tc.ThreadLoop));
    try {
      table.Add(getKeyIdFromParams(o,id),tc); }
    catch (Exception) {
      SError("Unfinished thread detected"+id); }
    t.Start();
  }

  public void SError(string s) {
    if (Error != null) Error(s);
  }

  private ContratTimeout.MyThreadContext extractFirst(bool anyway) {
    ContratTimeout.MyThreadContext tc = null;
    bool first = true;
```

```
      object key = null;
      foreach(object tt in table.Keys) {
        if (first && (anyway || !last_start_event_id.Equals(((MyThreadContext)table[tt]
          tc = (MyThreadContext) table[tt];
          key = tt;
          first = false;
        }
      }
      if (key != null) table.Remove(key); else return null;
      return tc;
    }

    public void Stop(params object[] o) {
      int id = Thread.CurrentThread.GetHashCode();
      DateTime endTime = DateTime.Now;

      MyThreadContext tc = (MyThreadContext) table[getKeyIdFromParams(o,id)];
      if (tc == null)
        if ((tc = extractFirst(true)) == null)
          return;

      string current_event_id = getKeyIdFromParams(o,tc.counter);
      if (last_start_event_id.Equals(current_event_id)) {
        if (table.Count >= 1) {
          if ((tc = extractFirst(false)) == null)
            return;
        } else table.Remove(id);
      }
      TimeSpan duration = endTime - tc.startTime;
      if ( duration.TotalMilliseconds <= timeout )
        tc.reset();
      /* event is sent by the thread */
    }

    public delegate void StringDel(string s);
    public event StringDel Error;
  }
}
```

Ce composant est assez complexe car il doit obtenir et sauvegarder les identifiants des requêtes. Cette information n'existant pas dans la plate-forme WComp, il l'extrait de l'identifiant du thread et des arguments de l'opération. Il crée un thread lors de l'appel à la méthode Start, qui émettra un événement pour notifier si le contrat est rempli ou violé dans le temps d'exécution imparti.

5.3.2 Version littérale avec propriété

Pour la deuxième version, nous voulons garder la propriété dans l'assemblage de composant qui appelle l'opération, c'est à dire dans le container qui utilise le service composite. Nous ajoutons alors deux sondes ainsi que des composants associés dans le service composite, qui nous fourniront les dates de début et de fin d'exécution de l'opération, et le calcul du temps d'exécution et sa sauvegarde seront fait en local, comme illustré dans la figure 5.8.
La différence avec la version sans propriété, est que dans ce deuxième cas, la propriété reste locale à l'application, alors que dans le premier cas, on est seulement notifié du dépassement du temps d'exécution fixé ou non. Il va donc falloir effectuer des modifications dans le service

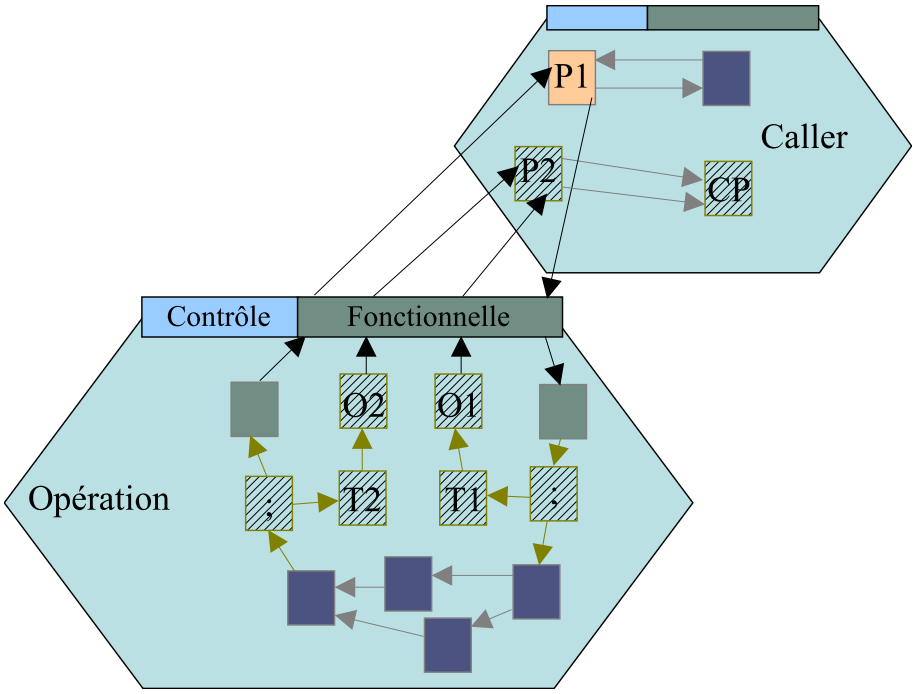


FIG. 5.8 – Illustration de la prise en charge du contrat Timer 1 dans WComp.

composite représentant l'opération, ainsi que dans celui qui appelle l'opération. Voici l'aspect d'assemblage qui sera tissé dans le service composite :
Pointcut :

```
sondeEntree:=/GetInfoProbe/
sondeSortie:=/GetInfoReturnProbe/
```

Advice :

```
schema ContratTimeout2Callee(sondeEntree,sondeSortie):
  timeGet1 : 'Wcomp.Beans.TimeGetter' ;
  timeGet2 : 'Wcomp.Beans.TimeGetter' ;
  sondeContrat1 : 'Wcomp.Probes.OutputProbe' ;
  sondeContrat2 : 'Wcomp.Probes.OutputProbe' ;

  sondeEntree.^Out -> (
    timeGet1.Check ; call
  )

  sondeSortie.In -> (
    timeGet2.Check ; call
  )

  timeGet1.^Time -> (
    sondeContrat1.In
  )

  timeGet2.^Time -> (
    sondeContrat2.In
  )
```

Le code du composant TimeGetter est assez simple, il envoie un événement contenant l'identifiant du thread et la date. Voici le code du composant à ajouter sur la plate-forme pour la prise en charge de ce contrat :

```
using System;
using WComp.Beans;
using WComp.EventedBeans;
using System.Threading;

namespace WComp.Beans
{
  [Bean(Category="Contract"]
  public class TimeGetter : EventedDrawable {
    public void Check(params object[] o) {
      int id = Thread.CurrentThread.GetHashCode();
      DateTime startTime = DateTime.Now;
      STime("Id:" + id + "\n" + startTime.ToString());
    }

    public void STime(string s) {
      if (Error != null) Time(s);
    }

    public delegate void StringDel(string s);
    public event StringDel Time;
  }
}
```

Lors de l'application de l'aspect d'assemblage ci-dessus, une nouvelle interface sera ajoutée au service composite. Les deux sondes ajoutées fourniront les temps de début et de fin d'exécution de l'opération. Nous utiliserons alors un nouveau composant proxy, éventuellement généré automatiquement par l'UPnPWizardDesigner qui communiquera avec cette nouvelle interface. L'ajout de ce composant proxy dans le container qui utilise le service composite permettra alors l'application de l'aspect d'assemblage suivant :
Pointcut :

```
proxyContrat:=/WcompFunctionalInterface_TimerContrat/
```

Advice :

```
schema ContratTimeout2Caller(proxyContrat):
  timeProp : 'WComp.Beans.TimeProperty' ;

  proxyContrat.^InputTime -> (
    timeProp.SetInputTime
  )

  proxyContrat.^OutputTime -> (
    timeProp.SetOutputTime
  )
```

Le code du composant TimeProperty est aussi assez simple. Il reçoit deux événements contenant les dates de début et de fin d'exécution, avec leur identifiant, et il calcule le temps écoulé entre les deux. Cette information restera disponible dans une propriété du composant.

```
using System;
using WComp.Beans;
using WComp.EventedBeans;
using System.Collections;

namespace WComp.Beans
{
```

```
  [Bean(Category="Contract")]
  public class TimeProperty : EventedDrawable {
    private Hashtable hStart = new Hashtable();
    private Hashtable hDelay = new Hashtable();

    public Hashtable ExecutionTime {
      get { return hDelay; }
    }
    public void SetInputTime(string s){
      string splitted = s.Split('\n');
      if (s.Length != 2)
        return;
      hStart.Add(s[0], s[1]);
    }

    public void SetOutputTime(string s) {
      string splitted = s.Split('\n');
      if (s.Length != 2)
        return;
      string startTime = hStart[s[0]];
      DateTime DTstart = System.DateTime.Parse(startTime);
      DateTime DTstop = System.DateTime.Parse(s[1]);
      hDelay.Add(s[0], ((TimeSpan)(DTstop - DTstart)).TotalMilliseconds.ToString());
    }
  }
}
```

5.3.3 Version en boîte noire

Pour WComp, l'implémentation en boîte noire est identique à celle sans propriété. Un composant contratTimeout recevra les événements OperationEntry et OperationExit, et calculera le temps d'exécution, et enverra un événement pour indiquer si la borne est dépassée. Le code sera donc identique, sauf que l'on suppose que le type du composant contratTimeout est déjà chargé dans le container.

5.3.4 coté client

L'implémentation du coté client fait intervenir les événements RequestSending et ResponseReceiving, qui dans WComp ne nécessitent pas d'intervenir à l'intérieur des boîtes noires, et donc de voir l'opération comme un service composite. Cet exemple pourra donc être réalisé simplement dans un assemblage de composants en lançant le chronomètre sur réception de deux événements. Voici le code de l'aspect d'assemblage correspondant :
Pointcut :

```
meteoProvider:=/MeteoService*/
```

Advice :

```
schema ContratTimeout2(meteoProvider):
  contratTimeout : 'Wcomp.Beans.ContratTimeout' ;
  textBox : 'System.Windows.Forms.TextBox' ;

  meteoProvider.GetInfo -> (
    contratTimeout.Start; call
  )

  meteoProvider.^GetMeteoReturn -> (
```

```
    if (contratTimeout.Check) {
      call
    } else {
      nop
    }
  )

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

5.4 ConFract - Fractal WS

Dans les livraisons 3.1 et 3.2 du projet FAROS, la prise en charge d'un certain nombre de contraintes a été considérée selon plusieurs points dans la plate-forme ConFract - Fractal WS :
- ConFract v1 ou v2 : la prise en compte des événements propres aux architectures est entièrement ouvert dans la version 2, alors qu'il est dédié à certains événements précablés dans la première version.
- Dans le cadre du procédé FAROS, les contraintes pourront être traduites dans le langage CCL-J (supporté par les deux versions de ConFract), transformées en code de surveillance en utilisant le *ServiceController* de ConFract v1, ou transformées en code spécifique à l'architecture de ConFract v2.

Il est aussi à noter que la partie concernant Fractal WS ne supporte nativement aucune forme de contractualisation, ni de mécanismes d'interception qui pourraient servir à leur mise en œuvre. La prise en compte des contraintes exprimées ne peut ainsi se faire qu'au niveau de la partie concernant ConFract.
Dans le cadre des réalisations effectuées dans le lot 4 du projet FAROS, ConFract v1 sera utilisé pour contractualiser l'application CIRE, démonstrateur fourni par EDF. Nous nous focaliserons ici sur un métamodèle de plate-forme correspondant à cette version de ConFract. Nous choisissons aussi d'effectuer les transformations vers un modèle du langage CCL-J, car les autres plates-formes ne fournissent pas un tel support de langage.

5.4.1 Métamodèle de plate-forme avec CCL-J

Nous illustrons le métamodèle de plate-forme CCL-J en reprenant le premier exemple développé en section 4.4, en partant de la variante littérale sans propriété, et en montrant le modèle résultant.
La figure 5.9 montre, de manière synthétique, quelles classes et informations seraient transformées pour obtenir des éléments du modèle CCL-J :
- Les informations de portée du contrat (association holder) sont utilisées pour déterminer le contexte de la spécification. Cela correspond à un objet OperationContext dans le modèle CCL-J.
- La forme de la clause et ses responsabilités guide la transformation vers une spécification sur une interface (responsabilité correspondante à une relation client/fournisseur entre appelant et appelé). Cela correspond à un objet InterfaceCCLJSpec dans le modèle CCL-J.
- L'analyse du LiteralTimeChecker et des deux observers qui lui sont associés va amener à la tranformation vers une seule postcondition (classe Post) regroupant les différentes expressions pour former une expression booléenne correspondante. Cette dernière partie de transformation est la plus complexe. Elle suppose qu'un parcours des observers a détecté des valeurs calculées sur des événements OperationEntry (comme beforeTime dans notre exemple), puis réutilisées sur des événements OperationExit. La reconnaissance de ce motif permet la production d'une expression résultante dans une postcondition, utilisant des

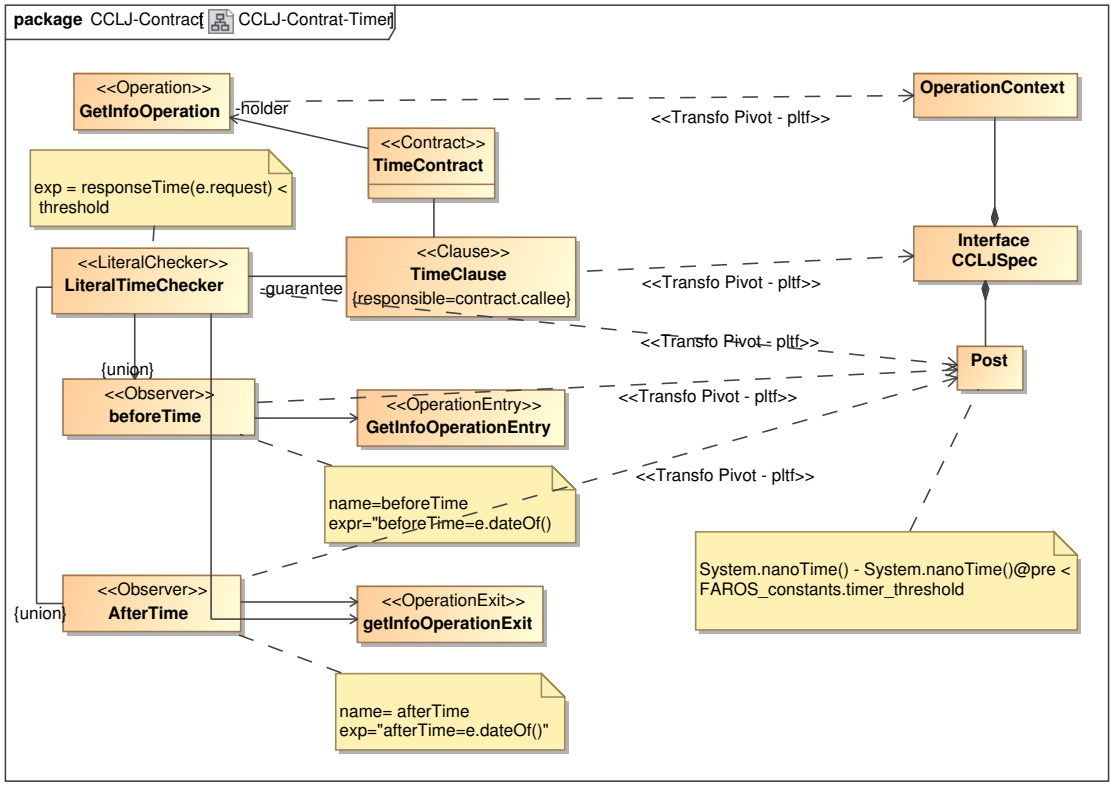


FIG. 5.9 – Transformation vers le modèle CCL-J du contrat Timer.

expressions suffixées @pre pour le calcul des valeurs observées à l'entrée de l'opération. Il est aussi à noter que nous considérons que les constantes (comme threshold) sont générées dans une classe utilitaire. Ceci pourra facilement évoluer lors de l'écriture des transformations.

– Enfin, la transformation vers un modèle CCL-J permet d'assurer que les vérifications se feront sur la même requête. Il n'est en effet pas utile à la transformation d'utiliser les informations de correspondance entre événement d'entrée et événement de sortie — correspondant à la même requête —, puisque c'est le système ConFract qui, en interprétant la spécification CCL-J générera lui-même les objets de contrat et d'observation avec les mêmes caractéristiques.

Le modèle ainsi obtenu permettra la production d'une spécification CCL-J de la forme suivante :

```
context Info i.getInfo()
  post: System.nanoTime() - System.nanoTime()@pre
          < FAROS_constants.timer_threshold;
```

Dans le cas d'un contrat exprimant la même contrainte, mais utilisant une propriété non fonctionnelle explicite (cf. paragraphe 4.4.2), le modèle résultant est censé être le même. Les propriétés non fonctionnelles doivent être simplement dépliées afin que l'expression de leur observation soit directement utilisée. Ainsi, une propriété non fonctionnelle observée ne sera plus explicitée dans le modèle CCL-J résultant.

Dans le cas d'un contrat en boîte noire (cf. paragraphe 4.4.3), la transformation vers le modèle est similaire au premier cas, comme le montre la figure 5.10. Les parties structurelles sont notamment identiques et seule l'expression correspond à l'appel de la méthode *check* sur le checker.

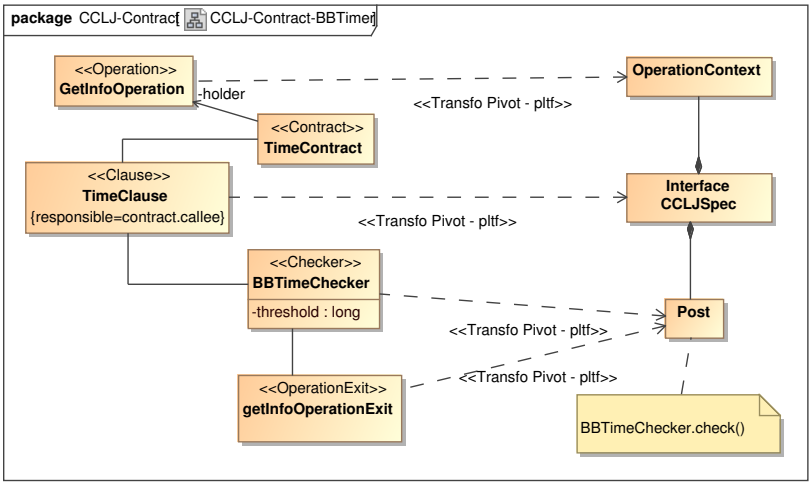


FIG. 5.10 – Transformation vers le modèle CCL-J du contrat Timer (boîte noire).

La figure 5.11 présente une vue générale du métamodèle de plate-forme dédié à l'expression des spécifications CCL-J.

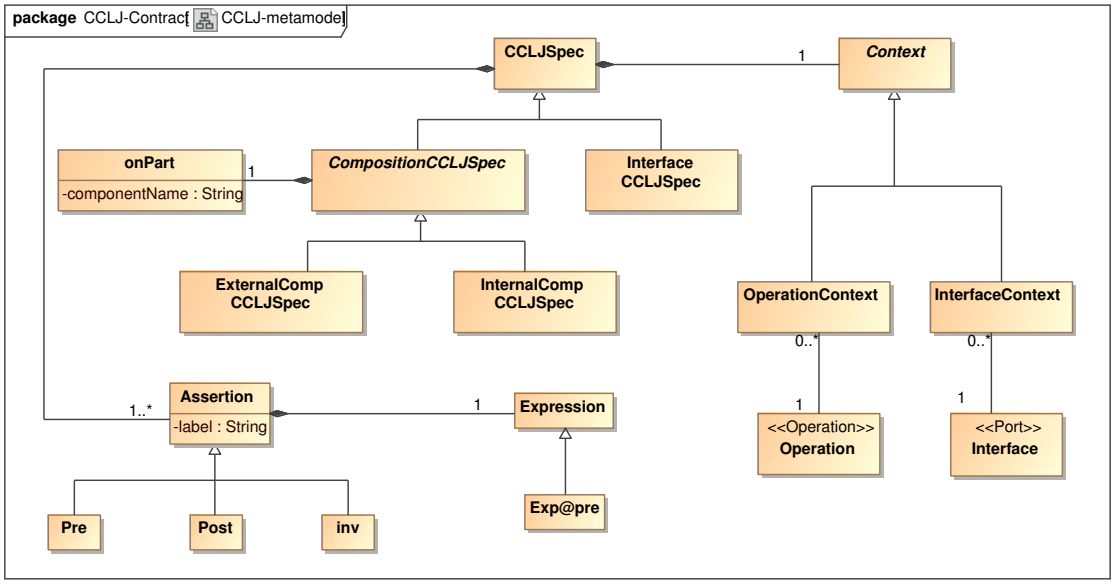


FIG. 5.11 – Métamodèle CCL-J.

Le métamodèle CCL-J décrit les spécifications sous la forme d'une hiérarchie de classes héritant de CCLJSpec. Trois classes feuilles représentent les trois formes de spécification possibles dans le langage. Quelle que soit cette forme, toute spécification repose sur un contexte (classe Contexte) qui peut être une opération, pour des assertions de type Pre ou Post, ou une interface pour des invariants. Ces types d'assertions héritent tous d'une classe Assertion, et chaque spécification peut contenir plusieurs assertions.
Nous effectuons maintenant une autre illustration en reprenant l'exemple de la section 4.5 qui vérifie que les requêtes envoyées sont bien authentifiées par un tiers de confiance. Nous nous appuyons sur la deuxième alternative de représentation, avec un modèle hiérarchique de composants, puisque c'est celle qui correspond le mieux à la famille de plate-forme dans

laquelle se trouve ConFract. Cet exemple nous permet d'illustrer la production d'une spécification de composants en CCL-J par le procédé FAROS. Cette spécification sera ensuite transformée en contrat de composition interne dans le système ConFract.

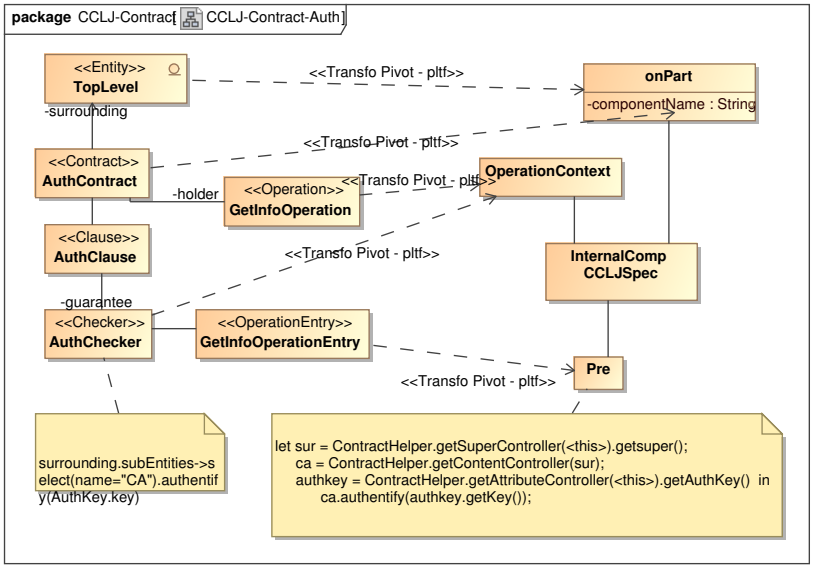


FIG. 5.12 – Transformation vers le modèle CCL-J du contrat d'authentification.

La figure 5.12 montre quelles classes et informations seraient transformées pour obtenir des éléments résultants du modèle CCL-J :

- Les informations de portée du contrat (association holder) et les responsabilités multiples du contrat sont utilisées pour déterminer la forme de la spécification, ici compositionnelle. Cela correspond aux objets OnPart, OperationContext et InternalCompCCLJSpec.
- La forme du checker, associée à un événement OperationEntry, aboutit à l'utilisation d'un objet Pre dans le modèle CCL-J.
- L'assertion résultante correspond à une transformation des navigations au sein du modèle UML pivot, vers des navigations réflexives au sein d'une architecture Fractal. Il est ainsi à noter qu'une partie des transformations devraient être réutilisables entre les modèles de plate-forme AoKell et ConFract, puisqu'ils correspondent à des implémentations du même modèle de composants Fractal. Dans le cas de ConFract, nous avons fait l'hypothèse, dans notre exemple, qu'une classe ContractHelper, était présente sur la plateforme afin de fournir des raccourcis d'accès aux contrôleurs de composants. Ces contrôleurs sont en effet les éléments qui vont assurer les fonctionnalités de navigation.

La spécification CCL-J produite aura la forme suivante :

```
on <topLevel> // surroundingComponentDeterminedByNavigation
 context getInfoProvidedPort.certificationMethod()
  pre :
    let sur = ContractHelper.getSuperController(<this>).getsuper();
        ca = ContractHelper.getContentController(sur);
        authkey = ContractHelper.getAttributeController(<this>).getAuthKey() in
       ca.authentify(authkey.getKey());
```

5.4.2 Modèle de plate-forme avec ConFract/Interact

La plate-forme ConFract/Interact fournit des moyens de prise en compte des spécifications CCL-J utilisées dans la section précédente. Par conséquent, les modèles et spécifications produites par transformation seront compatibles avec cette version de la plate-forme ConFract.
En revanche, afin de prendre en compte les événements, plus nombreux, interceptables dans la plate-forme ConFract/Interact, il va être nécessaire de produire un métamodèle de cette plate-forme. Comme cette dernière n'est pas directement utilisée dans les applications de validation, la production de ce métamodèle est décalée afin de profiter des premières mises en œuvre des transformations.

5.5 AOKell - FAC

Cette section illustre la prise en compte, dans la plate-forme AOKell-FAC, de la syntaxe de surface du chapitre 4.
Comme présenté dans les livrables du lot 3, la plate-forme AOKell-FAC offre un modèle de programmation unifié à base d'aspects et de composants. La plate-forme est décrite respectivement dans [SPDC06] (AOKell) et [PSDC06] (FAC). Elle a fait l'objet de la thèse de Nicolas Pessemier [PSDC06] qui s'est déroulée en collaboration entre l'équipe LIFL GOAL et France Telecom R&D. Nous renvoyons le lecteur aux livrables 3.1 et 3.2 pour une description détaillée de la plate-forme AOKell-FAC.
Nous proposons dans la suite de cette section, les patrons de code permettant de prendre en charge l'exigence de respect d'un délai donné en fonction des trois modalités définies dans la section 4.4. Avant cela, nous proposons quelques considérations générales, valables dans les trois cas, sur la façon dont la syntaxe de surface du chapitre 4 est prise en compte dans la plate-forme AOKell-FAC.

Prise en compte de la syntaxe de surface dans AOKell-FAC Les clauses `when` des checkers, des observers et des propriétés se traduisent par des coupes sur l'architecture métier. Les actions, clauses `do` des observers et des propriétés et clauses `check` des checkers se traduisent par des composants d'aspects. Les clauses `need` se traduisent par des dépendances entre composants d'aspects.

Version littérale sans propriété Les deux observers et le checkers se traduisent chacun par une coupe et un composant d'aspect. Les composants d'aspect correspondant aux deux observers fournissent chacun une interface qui est requise par le composant d'aspect correspondant au checker et qui permettent de récupérer le résultat des observers.

```
<!-- Observer beforeTime                          -->
<!-- coupe, composant d'aspect, interface fournie -->
<!-- implantation du composant d'aspect           -->

<weave root="this" ac="beforeTimeAC" aDomain="beforeTimeDomain"
  pointcutExp="EXECUTION c;i;o" />

<component name="beforeTimeAC">
  <interface name="aspectComponent" role="server"
    signature="org.objectweb.fractal.fac.api.AspectComponent" />
  <interface name="result" role="server" signature="Result" />
  <content class="BeforeTimeAC" />
</component>
```

```
interface Result {
  long getResult();
}

public class BeforeTimeAC implements AspectComponent, Result {
  long result;
  public Object invoke( FcInvocationMethod m ) throws Throwable {
    result = System.currentTimeMillis();
    return m.proceed();
  }
  public long getResult() { return result; }
}

<!-- Observer afterTime                                          -->
<!-- idem observer beforeTime                                    -->
<!-- à l'exception de l'implémentation du composant d'aspect -->

public class AfterTimeAC implements AspectComponent, Result {
  long result;
  public Object invoke( FcInvocationMethod m ) throws Throwable {
    Object ret = m.proceed();
    result = System.currentTimeMillis();
    return ret;
  }
  public long getResult() { return result; }
}

<!-- Checker LiteralTimeChecker          -->
<!-- coupe, composant d'aspect, liaisons -->
<!-- implantation du composant d'aspect  -->

<weave root="this" ac="literalTimeCheckerAC" aDomain="LiteralTimeCheckerDomain"
  pointcutExp="EXECUTION c;i;o" />

<component name="literalTimeCheckerAC">
  <interface name="aspectComponent" role="server"
    signature="org.objectweb.fractal.fac.api.AspectComponent" />
  <interface name="resultBeforeTime" role="client" signature="Result" />
  <interface name="resultAfterTime" role="client" signature="Result" />
  <content class="LiteralTimeCheckerAC" />
</component>

<binding client="literalTimeCheckerAC.resultBeforeTime"
         server="beforeTimeAC.result" />
<binding client="literalTimeCheckerAC.resultAfterTime"
         server="afterTimeAC.result" />

public class LiteralTimeCheckerAC implements AspectComponent {
  final static public long THRESHOLD = 1000;
  public Object invoke( FcInvocationMethod m ) throws Throwable {
    Object ret = m.proceed();
    if( resultAfterTime - resultBeforeTime > THRESHOLD ) {
      throw new ConstraintException("Délai d'exécution dépassé");
    }
    return ret;
  }
  @Requires private Result resultBeforeTime;
  @Requires private Result resultAfterTime;
```

```
}
```

Remarque 1 : nous utilisons dans la classe LiteralTimeCheckerAC la notation Fraclet `@Requires` pour indiquer une interface requise. Celle-ci peut être remplacée par une utilisation "classique" en Fractal de l'implantation de l'interface `BindingController`.
Remarque 2 : une information supplémentaire est nécessaire pour assurer le bon fonctionnement de cette mise en œuvre. En effet, les trois composants d'aspect `beforeTimeAC`, `afterTimeAC` et `literalTimeCheckerAC` interviennent avant ou après le même point de jonction qui correspond à l'opération dont on veut mesurer le délai entre le temps d'entrée et de sortie. De ce fait, il est nécessaire que ces trois composants d'aspect soient placés dans l'ordre suivant : `beforeTimeAC`, `literalTimeCheckerAC` et `afterTimeAC` (`literalTimeCheckerAC` est bien en deuxième position et non pas en dernière : les opérations après de `literalTimeCheckerAC` et de `afterTimeAC` s'exécutent dans l'ordre inverse de leur déclaration). Avec la plate-forme FAC-AOKell, cet ordre doit être fixé via le contrôleur d'aspect du composant métier sur lequel s'applique la contrainte.

Version littérale avec propriété Dans cette version, on retrouve les trois composants d'aspects correspondant aux deux observers et au checker du cas précédent. Notons néanmoins que l'implantation du checker est modifiée et qu'elle utilise maintenant la propriété. Cette dernière est quant à elle implantée par un composant d'aspect et fournit une interface qui est requise par le composant d'aspect correspondant au checker.

```
<!-- Observer observeProperty          -->
<!-- coupe, composant d'aspect, liaisons -->
<!-- implantation du composant d'aspect  -->

<weave root="this" ac="observePropertyAC" aDomain="observePropertyDomain"
  pointcutExp="EXECUTION c;i;o" />

<component name="observePropertyAC">
  <interface name="aspectComponent" role="server"
    signature="org.objectweb.fractal.fac.api.AspectComponent" />
  <interface name="responseTime" role="server" signature="Result" />
  <interface name="resultBeforeTime" role="client" signature="Result" />
  <interface name="resultAfterTime" role="client" signature="Result" />
  <content class="ObservePropertyAC" />
</component>

<binding client="observePropertyAC.resultBeforeTime"
         server="beforeTimeAC.result" />
<binding client="observePropertyAC.resultAfterTime"
         server="afterTimeAC.result" />

public class ObservePropertyAC implements AspectComponent, Result {
  private long result;
  public Object invoke( FcInvocationMethod m ) throws Throwable {
    Object ret = m.proceed();
    result = resultAfterTime - resultBeforeTime;
    return ret;
  }
  public long getResult() { return result; }
  @Requires private Result resultBeforeTime;
  @Requires private Result resultAfterTime;
}
```

```
<!-- Checker LiteralTimeChecker          -->
<!-- coupe, composant d'aspect, liaisons -->
<!-- implantation du composant d'aspect  -->

<weave root="this" ac="literalTimeCheckerAC" aDomain="LiteralTimeCheckerDomain"
  pointcutExp="EXECUTION c;i;o" />

<component name="literalTimeCheckerAC">
  <interface name="aspectComponent" role="server"
    signature="org.objectweb.fractal.fac.api.AspectComponent" />
  <interface name="responseTime" role="client" signature="Result" />
  <content class="LiteralTimeCheckerAC" />
</component>

<binding client="literalTimeCheckerAC.responseTime"
         server="observePropertyAC.responseTime" />

public class LiteralTimeCheckerAC implements AspectComponent {
  final static public long THRESHOLD = 1000;
  public Object invoke( FcInvocationMethod m ) throws Throwable {
    Object ret = m.proceed();
    if( responseTime > THRESHOLD ) {
      throw new ConstraintException("Délai d'exécution dépassé");
    }
    return ret;
  }
  @Requires private Result responseTime;
}
```

Remarque : même remarque que précédemment sur l'ordre des composants d'aspects qui doit être ici `beforeTimeAC`, `literalTimeCheckerAC`, `observePropertyAC` et `afterTimeAC`.

Version en boîte noire La version en boîte noire comporte un seul composant d'aspect `bBTimeCheckerAC` qui se charge de mettre en œuvre la contrainte.

```
<!-- Checker bBTimeChecker               -->
<!-- coupe, composant d'aspect, liaisons -->
<!-- implantation du composant d'aspect  -->

<weave root="this" ac="bBTimeCheckerAC" aDomain="bBTimeCheckerDomain"
  pointcutExp="EXECUTION c;i;o" />

<component name="bBTimeCheckerAC">
  <interface name="aspectComponent" role="server"
    signature="org.objectweb.fractal.fac.api.AspectComponent" />
  <content class="BBTimeCheckerAC" />
</component>

public class BBTimeCheckerAC implements AspectComponent {
  public Object invoke( FcInvocationMethod m ) throws Throwable {
    // Implémentation de la contrainte
  }
}
```

5.6 ORQOS

Cette section illustre la prise en compte, dans la plate-forme ORQOS, de la syntaxe de surface du chapitre 4. Comme présenté dans les livrables du lot 3 et la thèse de Fabien Baligand [Bal08], la plate-forme ORQOS collabore avec un moteur BPEL afin de gérer statiquement et dynamiquement la Qualité de Service (QoS) dans les orchestrations de services. Pour cela, elle suit les spécifications de politiques rédigées à l'aide du langage QoSL4BP dédié au domaine de la gestion de la QoS dans les compositions de services. Le langage QoSL4BP permet à un architecte de contrôler la QoS du point de vue de l'orchestration de services.

– Son approche déclarative permet de gérer implicitement les évènements. Ceci facilite l'écriture des transformations pour lesquelles les contraintes exprimées sur le pivot FAROS correspondent à des primitives du langage QoSL4BP mais rend par contre difficile, voire impossible, la prise en compte de contraintes spécifiques, comme les contraintes applicatives proposées dans le chapitre 4. Il gère ainsi implicitement les évènements entrants ou sortants de l'orchestration BPEL (notés *RequestSending, ResponseReceiving, OperationEntry et OperationExit* dans la table 5.1) mais de façon limitée ou indirecte les évènements liés au cycle de vie d'une application. Il gère par contre des évènements non couverts ici comme l'entrée ou la sortie d'une activité composite.
– Il couvre des fonctionalités non couvertes par FAROS, comme la réaction aux violations des contrats (e.g. pour appliquer une stratégie de replanification des services invoqués) mais ne prend en compte que des critères de QoS génériques (durée, coût, sécurité, etc.).

Nous illustrons tout d'abord la mise en oeuvre par ORQOS du contrat portant sur une exigence de respect d'un délai lors d'un appel de service (cf. chapitre 4.4.4). La politique cible dans le langage QoSL4BP est décrite dans la figure 5.13.

```
POLICY "TimeContractRq" = {
  SCOPE = { INVOKE[<Activity>] }
  INIT = { SET_REQUIREMENT([RESPONSETIME < <threshold>]) }
  RULE = {
    VIOLATION_PROVIDER(SCOPE) ->
      THROW("Délai non respecté par le service appelé par <Activity>");
  }
}
```

FIG. 5.13 – Politique de vérification d'une contrainte de temps (exigence)

Comme toute politique QoSL4BP, celle-ci regroupe dans un même module trois sections :

– SCOPE permet de faire le lien entre l'activité cible (éventuellement composite) de l'orchestration et la portée des spécifications de QoS de la politique. Le SCOPE correspond ici, dans le modèle pivot, à l'activité de l'entité "caller" qui invoque le service sur lequel porte le contrat (cf. chapitre 4.4.4). Les évènements *RequestSending, ResponseReceiving* du contrat sont implicitement gérés par ce SCOPE réduit ici à une simple activité *invoke*.
– INIT couvre l'objectif de QoS statique associé à ce SCOPE. La primitive `SET_REQUIREMENT` correspond ici à l'exigence du délai maximum `<threshold>` exprimée par l'entité "caller" du contrat.
– RULES couvre la partie dynamique de gestion de la QoS associée à ce SCOPE. Cette notion n'étant pas couverte par le procédé FAROS, elle est ici spécifiée par défaut par une remontée d'exception indiquant que le service appelé par `<Activity>` a violé sa garantie.

Nous illustrons maintenant la mise en oeuvre par ORQOS du contrat symétrique, portant sur une offre de respect d'un délai coté serveur (cf. chapitre 4.4.1). La politique cible dans le

langage QoSL4BP est décrite dans la figure 5.14.

```
POLICY "TimeContract" = {
  SCOPE = { PROCESS }
  INIT = { SET_OFFER([RESPONSETIME < <threshold>]) }
  RULE = {
    VIOLATION_ACTIVITY(SCOPE) ->
      THROW("Délai non respecté par le processus");
  }
}
```

FIG. 5.14 – Politique de vérification d'une contrainte de temps (offre)

Les différences par rapport à la politique précédente sont :
– Le SCOPE correspond dans le modèle pivot à l'activité qui implante le service *getInfo*. Cette activité correspond au processus BPEL géré par ORQOS, elle est notée PROCESS dans la syntaxe QoSL4BP. Les évènements *OperationEntry, OperationExit* du contrat sont implicitement gérés par ce SCOPE.
– la section INIT utilise la primitive `SET_OFFER` pour traduire l'offre d'un délai maximum `<threshold>` sur la durée du processus, garantie par l'entité "callee" du contrat.