République Algérienne Démocratique Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université d'Ibn Khaldoun – Tiaret

Faculté des Mathématiques et de l'Informatique

Département Informatique

Thème

Sémantique Formelle du BPEL avec K Framework

Pour l'obtention du diplôme de Master II

Spécialité : Génie Informatique

Option : Systèmes Embarqués et Temps Réel

Rédigé par : ANNANE Khaled

Dirigé par : BEKKI Khadhir

Année Universitaire 2013-2014

رَبَّنَا تَقَبَّلْ مِنَّا إِنَّكَ أَنتَ السَّمِيعُ الْعَلِيمُ

إلى الوالدين الكريمين سائلا المولى العلي القدير أن يكونوا من أهل الآية الكريمة :

أدخلوها بسلام آمنين

إلى كل من علمني حرفا إلى أساتذتي الكرام وبالأخص الأستاذ بكي خثير, راجيا المولى العزيز الحكيم أن يكونوا ممن قال فيهم المصطفى عليه أفضل الصلاة و أزكى التسليم :

من سلك طريقا يلتمس فيه علما سهل الله له به طريقا إلى الجنة

إلى كل من ساهم في هذا الجهد المتواضع من قريب أو من بعيد أسأل الله العلي العظيم أن يكونوا ممن قال فيهم رسول الله عليه الصلاة و السلام :

والله في عون العبد ما كان العبد في عون أخيه

إلى كل العائلة

* * *

وصلى الله على نبينا محمد ﷺ , ما ذكره الذاكرون وغفل عنه الغافلون

Résumé

Définir une sémantique formelle d'un langage de programmation permet de comprendre le comportement des programmes écrits dans ce langage, ainsi de détecter les erreurs de dysfonctionnement. Cette étape est généralement ignorée et considéré par beaucoup, malheureusement même des programmeurs experts trop coûteux et inutilisable [42]. Notre travail consiste à utiliser le Framework K, un outil de définition et d'analyse des langages de programmation basé sur la logique de réécriture et le système Maude, pour définir la sémantique formelle du langage BPEL.

Mots-clés : BPEL, Logique de réécriture, Maude, K Framework

Table des matières

Listes des Figures

Introduction Générale

Contexte générale :

Un processus métier est un ensemble d'activités de forte granularité dont chacune réalise une fonctionnalité bien précise en consommant et en produisant des données. Ces activités interagissent avec l'environnement du processus et s'exécutent dans un ordre spécifique.

Les entreprises veulent que leurs partenaires puissent accéder directement à leurs fonctionnalités vu l'évolution qu'a connue le réseau internet. Pour ce faire, il doit y avoir une intégration entre les différents systèmes d'information: cette intégration est assurée par les services Web qui permettent de relier des processus métiers à l'aide de protocoles d'Internet standards, c'est-à-dire, ces entreprises voient les Services Web comme étant des interfaces abstraites et standardisées de leurs processus d'affaire.

L'utilisation des méthodes formelles pour la vérification de Services Web s'est avérée précieuse au sein de la dernière décennie. En fait, les méthodes formelles ont été la pierre angulaire et la base mathématique qui manquait dans le monde des services web.

Les méthodes formelles ont été utilisées afin d'élever le niveau de confiance des utilisateurs dans les applications web en particulier en raison des grandes quantités des mouvements monétaires qui transitent tous les jours via le web. La sémantique formelle pour les langages de composition de services, en particulier pour BPEL, ont été intensivement étudiés résultant en plusieurs ouvrages. Des approches fondées sur les réseaux de Petri, les algèbres de processus et les automates ont été proposées. Une transformation de descriptions informelles à des modèles formels a donc été introduite.

Problématique :

Vu la concurrence, les entreprises aujourd'hui nécessitent d'appuyer sur des processus métiers correctes, fiables et vérifiées. BPEL se présente aujourd'hui comme un langage standard pour la composition et l'exécution des processus métiers. D'où il nécessite une vérification minutieuse avant son déploiement. Plusieurs travaux ont été donnés pour la vérification formelle du BPEL. Mais la plupart d'eux se basent sur la transformation du processus BPEL vers un autre formalisme (par ex RDP) pour la vérification formelle. Ce qui influe sur la préservation de la sémantique du code BPEL ainsi sur la qualité de la vérification.

Trouver d'autres façons de vérification formelle basant sur la syntaxe du BPEL s'avère nécessaire et prometteuse.

Contribution :

Avoir un seul cadre sémantique contenant tous les outils nécessaire pour exécuter et analyser les programmes écrit dans un langage (Analyseurs, interpréteurs, compilateurs, vérificateurs de modèles…) en facilitant la définition des langages de programmation quel que soit leurs complexité est l'un des anciens rêves de la communauté des langages de programmation, le Framework K développé récemment, vise à apporter une sémantique formelle grand publique, en fournissant une notation intuitive et un ensemble d'outils d'analyses et de vérification qui peuvent être utilisés avec n'importe quel langage.

Le Framework K a été utilisé pour formaliser plusieurs langages par exemple C [23], Python [24], Java [25] et d'autre, notre contribution est d'utiliser ce Framework pour développer la sémantique formelle du BPEL.

Organisation :

L'objectif du premier chapitre de ce mémoire est de présenter le vocabulaire et la grammaire de langages BPEL.

Le deuxième chapitre sera dédié à la présentation des principaux concepts du formalisme de la logique de réécriture, du système Maude.

Le troisième chapitre est une introduction au Framework K en approchant les principes de base, ainsi les différents travaux de recherche concernant la vérification de processus BPEL.

Le dernier chapitre présente la structure et les composants majeurs de la sémantique du langage BPEL avec K Framework.

Chapitre 1 : Le Langage BPEL

Introduction

Aujourd'hui, les entreprises ont besoin de s'adapter rapidement aux nouvelles approches proposant des services meilleurs, rapides et complets. Des plates-formes communes d'échange et de collaboration d'information constituent un élément clé pour satisfaire ces besoins compte tenu leur implémentation facilitée par l'utilisation de l'architecture orientée services (SOA).

Les services les plus utilisés dans ces plates-formes sont des services web. Les services web ont gagné beaucoup d'importance dans le développement des logiciels grâce à leur indépendance par rapport aux langages de programmation et parce qu'ils permettent d'être réutilisables pour fournir des services plus élaborés.

Pour réaliser une application basée sur les services web, il nous faut un plan prédéfini permettant de décrire la coordination des services. Ce plan s'appelle l'Orchestration. L'Orchestration décrit un modèle de coordination de services qui sera exécuté à l'aide d'un Moteur d'Orchestration.

L'Orchestration est représentée par un *workflow* (flux de travail) coordonnant des services (et plus particulièrement des services web), ce qui offre des avantages comme l'optimisation et l'automatisation de tâches. Un *workflow* modélise la gestion informatique de l'ensemble des tâches à accomplir par les différents acteurs impliqués dans la réalisation d'un processus métier. Ces acteurs peuvent être des humains réalisant les tâches demandées, ou des services (services web, applications locales ou distantes...). Le *workflow* se charge de l'administration ou le contrôle de la coordination de ces acteurs et services.

Comme toute modélisation d'un processus métier, l'orchestration des services web nécessite l'élaboration d'un langage. Plusieurs langages ont étaient développé dans ce but : WSFL (*Web Service Flow Language*) développé par IBM et étant une extension de FL, leur langage de modélisation de processus ; et XLANG de Microsoft qui était une extension de WSDL [1] (web *service description language),* un langage qui décrit les web services. La fusion de ces deux langages a donné naissance à un standard base sur le langage XML. Ce langage s'appelle BPEL *(business process execution language).*

En 2003, la collaboration entre IBM et Microsoft a mené à la publication de la norme BPEL4WS 1.1 [2] (BPEL for Web Services) ; celle-ci a ensuite été soumise au consortium OASIS, et ce nouveau travail a abouti en 2007 à la norme WSBPEL 2.0 [3] qui apporte quelques nouveautés. Le terme « BPEL » désigne de manière globale ces deux langages. Dans le cadre de notre chapitre, nous présentons les concepts de base de ce langage avec quelques exemples et ce, depuis la dernière version de la spécification de ce langage [3].

I. Structure des processus BPEL

Un processus BPEL est un container dans lequel la relation avec des partenaires externes, des déclarations pour les données du processus, des gestionnaires pour des buts divers et plus important, des activités à exécuter, sont introduits. Le container du processus comporte deux attributs (« name » et une déclaration de l'espace de nom) comme le montre l'exemple suivant sur la figure 1.1 :

```
<process name="Process"
   targetNamespace="http://oasis-open.org/WSBPEL"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable" />
```

Figure 1.1. Processus BPEL

La déclaration de l'espace de nom signifie que ce processus est exécutable. Un espace de nom additionnel est prévu pour les processus abstraits. Les processus abstraits décrivent partiellement le comportement des processus sans se préoccupe des détails lies à l'exécution. Ce comportement expose (de manière typique) un modèle de processus ou un comportement externe visible du processus à l'égard des partenaires métiers sans montrer sa logique interne. Le processus exécutable, par contre, définit le comportement métier complet des parties internes et externes. L'élément « process » est le container le plus à l'extérieur. En effet, tout partenaire, données de processus ou gestionnaires déclarer à l'intérieur peut être considère comme global. BPEL support également un mécanisme de déclaration de toutes ces propriétés de manière local. L'élément structurant qui assure cette fonctionnalité est « scope ».

Avec l'aide des scopes, il est possible de diviser un processus métier chacun contenant une portion de la logique métier globale. Par exemple, les données du processus déclarées localement dans un scope ne sont pas visibles à l'extérieur de ce dernier (seulement valide à l'intérieur d'une partie spécifique du processus).

II. Relation entre BPEL et ses partenaires

Les processus métiers BPEL offrent la possibilité d'agrégation des services web et définissent la logique métier pour chaque interaction entre services. Il s'agit d'une orchestration des interactions entre services web. Chaque interaction peut être vue comme une communication avec un partenaire métier. Une interaction est décrite à l'aide de liens entre partenaires *(partner links)*. Ce sont des instances spécifiant des connecteurs types qui spécifient les types de ports WSDL se trouvant à chaque extrémité.

Pour un partenaire, il peut y avoir un ensemble de liens. Un lien peut être vu comme un canal de communication. Une telle interaction a deux côtes : le processus invoque le partenaire et vice versa. Ainsi, chaque *partnerlink* est caractérisé par un type (*partnerLinkType*) et un rôle. Cette information identifie la fonctionnalité qui doit être fournit par le processus métier et le service partenaire (voir la figure 1.2).

```
<partnerLinks>
  <partnerLink name="ClientStartUpLink"
     partnerLinkType="wsdl:ClientStartUpPLT"myRole="Client"/>
</partnerLinks>
```

Figure 1.2. Interaction entre un processus et son partenaire

Les déclarations d'un *Partner link* peuvent prendre place directement à l'intérieur de l'élément «*process* » pour permettre leur accessibilité par les constructions BPEL à l'intérieur du *process,* ou à l'intérieur d'un élément « scope » pour une accessibilité exclusive par les éléments fils du « scope ».

III. Etat d'un processus BPEL

Les données d'un processus représentent l'ensemble des variables déclarées dans le processus ou dans un « scope » à l'intérieur de ce processus. Les variables contiennent les données qui constituent l'état du processus métier durant son exécution. De plus, Les données peuvent être écrit sur ces variables ou lu à partir de ces derniers. Les valeurs contenues dans ces variables peuvent provenir de deux sources : soit à partir des messages échangées avec un partenaire, soit à partir des données intermédiaires qui sont propres au processus (des données privées). Pour respecter la convention de type avec le partenaire, toutes les variables dans BPEL doivent être de type message WSDL, de type XML schéma simple, de type XML schéma complexe ou des éléments XML schéma. Afin de changer l'état d'un processus à travers le changement du contenu de ses variables, un langage d'expression pour manipuler et interroger les variables est nécessaire. En BPEL,

le langage par défaut prévu pour cet effet est Xpath 1.0 [4].

Il est possible de déclarer une variable par un nom et l'un des trois types mentionnes plus haut (avec les trois attributs : « type », « messageType » ou « element ») (voir la figure 1.3).

```
<variables>
   <variable name="myVar1" messageType="myNS:myWSDLMessageDataType" />
   <variable name="myVar1" element="myNS:myXMLElement" />
   <variable name="myVar2" type="xsd:string" />
   <variable name="myVar2" type="myNS:myComplexType" />
</variables>
```

Figure 1.3: Variables BPEL

Les déclarations de variables peuvent apparaitre directement à l'intérieur des balises du « process », et ils sont ainsi visibles aux constructions contenues dans ces balises. Ils peuvent apparaitre également à l'intérieur d'un « scope » (ils sont ainsi visibles exclusivement aux fils du « scope »).

IV. Comportement d'un processus BPEL

La majorité des structures d'un processus BPEL sont des activités. Il y a deux types Activités structurées (peuvent contenir d'autres activités et définir la logique métier entre eux) et Activités de base (ne peuvent pas contenir d'autre activités).

1. Fournir et consommer des services web

En BPEL, il existe quelques activités simples qui ont pour but de fournir et consommer des messages à travers leur interaction avec les services web partenaires. Ces activités sont :« receive» (reception), « reply » (reponse) et « invoke » (appel ou invocation). Toutes ces activités permettent l'échange des messages avec des partenaires externes (services). L'intérêt de l'activité « receive » est de recevoir les messages depuis ses partenaires externes. Par conséquent, ce type d'activité spécifie toujours les *Partner links* et l'opération des partenaires.

Des variables sont nécessaires également pour contenir les données essentielles reçus d'un partenaire. Une activité de type réception peut avoir un « reply » associe si elle est utilisée pour fournir une opération WSDL de type requête-réponse (voir la figure 1.4).

```
<receive name="ReceiveRequestFromPartner"
   createInstance="yes"
   partnerLink="ClientStartUpPLT"
   operation="StartProcess" ... />
```

Figure 1.4: Activité « receive »

L'attribut « createInstance » à l'intérieur de l'activité «receive» signifie qu'il est possible d'utiliser une activité de type « receive » avec une valeur «yes» pour l'attribut pour créer une nouvelle instance du processus. Par contre, si cette valeur est « no » alors le message reçu sera consommé par l'instance déjà exécutée. Comme déjà mentionné, une activité de type «receive» peut avoir une activité « reply » associée. Par exemple, si un client veut commander un livre depuis un processus de vente. Un client enverrait une requête à l'activité de type « receive » du processus de vente. Le processus ferait quelques vérifications d'ordre logique (internes) (par exemple si le livre est disponible...). Ensuite, il serait naturel pour le processus de répondre au client pour le tenir informer à propos du succès ou de l'échec de sa commande (voir la figure 1.5).

```
<reply name="ReplyResponseToPartner"
  partnerLink="ClientStartUpPLT"
  operation="StartProcess" ... />
```

Figure 1.5. Activite « reply »

La troisième activité liée aux services web est « invoke ». C'est une activité utilisée pour appeler un service web fournit par un partenaire. Le « partnerLink » et l'opération du service web doivent être spécifiés (voir la figure 1.6).

```
<invoke name="InvokePartnerWebService"
  partnerLink="BusinessPartnerServiceLink"
  operation="partnerOperation" ... />
```

Figure 1.6. Activité « invoke »

2. Structuration de la logique des processus

BPEL fournit des moyens pour structurer la logique métier suivant les besoins. Si les activités ont besoin d'être exécutés dans un ordre séquentiel alors l'activité « sequence » est utilisée (voir la figure 1.7).

```
<sequence name="InvertMessageOrder">
 <receive name="receiveOrder" ... />
 <invoke name="checkPayment" ... />
 <invoke name="shippingService" ... />
 <reply name="sendConfirmation" ... />
</sequence>
```

Figure 1.7 Activité « sequence »

Une autre activité utilisée pour structurer la logique métier est l'activité « if-else » (si-sinon). La construction peut être devinée depuis les langages de programmation traditionnels. L'activité « if-else » permet la sélection d'une possibilité parmi plusieurs.

Comme avec toutes les expressions de BPEL, Xpath peut être utilisé pour formuler la condition (voir la figure 1.8).

```
<if name="isOrderBiggerThan5000Dollars">
 <condition>
    <!--  XPATH expression  -->
    $order &gt; 5000
 </condition>
 <invoke name="calculateTenPercentDiscount" ... />
<elseif>
 <condition>
    <! --  XPATH expression  -->
    $order &gt; 2500
 </condition>
 <invoke name="calculateFivePercentDiscount" ... />
</elseif>
 <else>
  <reply name="sendNoDiscountInformation" ... />
 </else>
</if>
```

Figure 1.8. Activité « if-else »

3. Activités répétitives

BPEL offre trois activités permettant une exécution répétitive d'une portion de logique métier. L'une de ses activités est l'activité « while ». L'activité « while » est une activité structurée (elle a un fils emboite à l'intérieur). Le « while » permet une exécution répétitive de l'activité emboitée aussi longtemps que la condition est vrai. Cette condition est placée au début de l'activité et elle est évaluée à chaque itération. Cela veut dire que le corps du« while » ne doit pas être exécuté (voir la figure 1.9).

```
<while>
 <condition>
  $iterations &gt; 3
 </condition>
 <invoke name="increaseIterationCounter" ... />
</while>
```

Figure 1.9. Activité « while »

Par contre, l'activité « repeatUntil » est différente par le fait que son corps est exécuté au moins une fois, parce que la condition est évaluée à la fin de chaque itération (voir la figure 1.10).

```
<repeatUntil>
  <invoke name="increaseIterationCounter" ... />
  <condition>
    $iterations &gt; 3
  </condition>
</repeatUntil>
```

Figure 1.10. Activité « repeatUntil »

La troisième activité du groupe d'activités répétitives est le « forEach ». Dans son comportement par défaut, l'activité « forEach » itère séquentiellement N fois sur un ensemble d'activités. Concrètement, le « forEach » itère son contenu (son « scope ») exactement N fois. N étant égale le « finalCounterValue » moins le « startCounterValue » plus 1 (voir la figure 1.11).

```
<forEach parallel="no" counterName="N" ...>
  <startCounterValue>1</startCounterValue>
  <finalCounterValue>5</finalCounterValue>
  <scope>
   <documentation>check availability of each item ordered</documentation>
   <invoke name="checkAvailability" ... />
  </scope>
</forEach>
```

Figure 1.11. Activité « forEach »

Deux variantes « forEach » existent : séquentielle et parallèle, comme spécifié par l'attribut « parallel ». Une restriction est appliquée au « forEach » : toutes les activités que nous venons de voir peuvent avoir une activité emboité à l'intérieur, ce qui n'est pas le cas du « forEach » qui ne peut contenir que des « scope ».

4. Traitement parallèle

Jusqu'ici, nous avons introduit des concepts variés sur la manière dont un processus métier peut être structuré séquentiellement. Toutefois, il est parfois préférable ou même nécessaire d'exécuter en parallèle. Pour cette raison, BPEL offre l'activité de type « flow ».Dans l'exemple suivant, un ensemble composé de trois activités *(checkFlight* (vérifier le vol), *checkHotel* (vérifier l'hotel) et *checkRentalCar* (vérifier la voiture de location)) sont exécutées en parallèle (leurs services web correspondants seront exécutés en concurrence) [5].Les trois activités démarrent en concurrence au démarrage du « flow » (voir la figure 1.12).

```
<flow ...>
   <links> ... </links>
   <documentation>
      check availability of a flight, hotel and rental car concurrently
   </documentation>
   <invoke name="checkFlight" ... />
   <invoke name="checkHotel" ... />
   <invoke name="checkRentalCar" ... />
</flow>
```

Figure 1.12: Activite « flow »

Les sémantiques des éléments *link* sont plus riches que celles de notre exemple. Un *link* peut avoir une condition de transition associée à lui et qui influence son statut. Si aucune condition de transition *(transitionCondition*) n'est spécifiée, le statut du *link* est vrai *(true)*.Par contre, si elle est spécifiée, elle fixe le statut du *link*. Voici un exemple de la figure 1.13 :

```
<flow ...>
   <links>
      <link name="request-to-approve" />
      <link name="request-to-decline" />
   </links>
   <receive name="ReceiveCreditRequest"
      createInstance="yes"
      partnerLink="creditRequestPLT"
      operation="creditRequest"
      variable="creditVariable">
      <sources>
         <source linkName="request-to-approve">
            <transitionCondition>
               $creditVariable/value &lt; 5000
            </transitionCondition>
         </source>
         <source linkName="request-to-decline">
            <transitionCondition>
               $creditVariable/value >= 5000
            </transitionCondition>
         </source>
     </sources>
 </receive>
 <invoke name="approveCredit" ...>
    <targets>
         <target linkName="request-to-approve" />
    </targets>
 </invoke>
 <invoke name="declineCredit" ...>
      <targets>
         <target linkName="request-to-decline" />
      </targets>
 </invoke>
</flow>
```

Figure 1.13. Activité « flow » avec « link » et « transitionCondition »

Le lien *request-to-approve* (demande-à-accord) possède une condition de transition qui vérifie si la partie valeur de la variable *creditVariable* a une valeur inférieure à 5000. Si c'est le cas, le statut du *link* de *request-to-approve* sera fixé à « true », sinon il sera fixé à « false ».Puisque la condition de transition du lien *request-to-decline* est exactement l'opposé (supérieur à 5000), alors l'une des deux activités qui succèdent *(approveCredit* et *declineCredit* sera exécutée).

Les conditions de transition offrent un mécanisme pour diviser le flux de contrôle en se basant sur certaines conditions. Par conséquent, un mécanisme de fusion doit aussi être offert. BPEL réalise ça avec les conditions de jointure *(joinCondition*). Les conditions

de jointure sont associées aux activités, habituellement si l'activité a des liens entrants.

Une *joincondition* spécifie pour une activité une sorte de condition de départ *(start condition)*. Par exemple, tous les liens entrants doivent avoir un statut fixé à « true » pour que l'activité s'exécute, ou bien au moins un seul lien entrant doit avoir un tel statut.

5. Manipulation des données

Dans les sections précédentes nous avons vue comment un processus reçoit les données métiers. Mais en pratique, ces données ont besoin parfois d'êtres divisées en plusieurs parties, ou jointent depuis plusieurs sources. Par exemple, un message entrant au processus contient les parties suivantes :

- Le nom du client
- Un nombre d'articles commandes
- Une adresse d'expédition
- Information de la carte de crédit
- Etc.

Une entreprise comprend un département chargé de gérer les expéditions et un autre qui gère la disponibilité. Un partenaire métier vérifie la validité de la carte de crédit. Il est évidant que ce service partenaire n'a rien à voir avec les commandes du client, il prend seulement en compte le nom du client et les informations de sa carte de crédit [5]. Dans ce scenario, nous avons besoin de : prendre deux informations faisant partie du message entrant du processus, créer un nouveau message qui contient juste ces deux parties et utiliser ce nouveau message comme message entrant au service *creditCardCheck* (vérification de la carte de crédit).Il devient alors nécessaire d'introduire un nouveau mécanisme pour la manipulation des données à l'intérieur d'un processus métier BPEL.

Pour ce fait, BPEL offre une activité d'affectation *(assign)*. L'activité *assign* contient une ou plusieurs opérations de copies *(copy*). Chacune possède « from-spec » et « to-spec »,indiquant la source où l'information est copiée et la destination où l' information est collée. Dans l'exemple suivant, le contenu de la variable est copié dans une autre variable ayant un type de données similaire (voir la figure 1.14) :

```
<assign>
   <copy>
      <from
      variable="TimesheetSubmissionFailedMessage" />
      <to variable="EmployeeNotificationMessage" />
   </copy>
</assign>
```

Figure 1.14. Activité « assign »

Toutefois, l'exemple textuel montre que ce n'est pas suffisant car souvent nous avons besoin de copier une partie du contenu de la variable dans une autre, ou même une partie du contenu de la variable dans une partie d'une autre variable. Ainsi, nous avons besoin d'un moyen pour référencer de telles parties. Un des buts de BPEL est de ne pas inventer d'autres langages XML pour la manipulation des données, mais de réutiliser d'autres standards tels que XPath et XSLT [4] pour cette fin .

6. Traitement des exceptions

Presque tous les concepts expliqués, à travers les exemples précédents, supposent que tout se passe bien durant l'exécution d'un processus métier. En réalité, un langage comme BPEL doit être capable de gérer les situations exceptionnelles (par exemple, appeler un web service non disponible). En d'autres termes, un mécanisme de gestion des exceptions doit être offert, et aussi un moyen de gérer ces situations doit être fourni.

BPEL offre ainsi le concept de gestion des erreurs (« fault handlers »). Un gestionnaire d'erreurs peut être attaché à un *scope,* un *process* ou même à travers des abréviations à l'intérieurs de l'activité *invoke.* Dans ce qui suit, seul le traitement des erreurs au niveau du *process* seront exposés. Un gestionnaire d'erreurs est installé dès le démarrage d'un *scope* associe à lui. Par exemple, le gestionnaire d'erreurs au niveau d'un processus est installé au démarrage du processus. Si le processus termine normalement, le gestionnaire d'erreurs est abandonné, par contre si une situation d'erreur se présente, elle est propagée au gestionnaire d'erreurs (voir 1.15).

```
<faultHandlers>
   <catch faultName="BookOutOfStockException"
    faultVariable="BookOutOfStockVariable">
    …
   </catch>
   <catchAll>...</catchAll>
</faultHandlers>
```

Figure 1.15. Gestionnaire d'erreur dans un processus

Dans cet exemple, un gestionnaire d'erreurs peut avoir deux types de fils : une construction ou plus de type *catch* (capture), et au plus un *catchall.* Chaque construction *catch* doit fournir une activité (dans l'endroit où se trouvent les points de suspension dans la figure précédente) qui exécute la gestion de l'événement pour un type d'erreur spécifique. Un appel de vérification de la disponibilité *(checkAvailability)* d'un service web pour la réservation, pour voir si le livre réservé est en stock, et la réponse du service web, peuvent se faire à travers le lancement d'un *BookOutOfStockExeption* (qui sera déclaré dans l'interface WSDL du service). Une construction *catch* possède des attributs additionnels : un *faultName* (le nom de l'erreur) et un *faultVariable.* Quand un *faultVariable* est utilisé, soit l'attribut *faultMessageType* ou *faultElement* doit être spécifié. Le *faultVariable* pointe localement sur une variable déclarée à l'intérieur d'une construction *catch* basée sur le *faultMessageType* ou le *faultElement.*

Optionnellement, le gestionnaire des erreurs peut finir avec une construction *catchall.* Le but est de fournir un moyen de gestion d'erreurs par défaut. Par exemple, si le service web *checkAvailability* ne lance pas seulement *BookOutOfStockExeption,* mais aussi un *BookOutOfPrintExeption* (exception pour un livre non imprimé) et *BookTitleNotFoundExeption* (exception pour un titre non trouvé). Si la distinction entre la gestion d'erreurs de ces deux derniers n'est pas voulue, il est possible de compléter le *catchAll* avec les activités de gestion d'exception appropriées.

V. Raffinement de la structure d'un processus

En BPEL, il est possible de structurer un processus d'entreprise dans une hiérarchie de *scopes.* Chaque *scope* peut avoir ses propres définitions de variables, liens partenaires, messages échangés, ensembles corrélés et gestionnaires. Cela limite la visibilité de ces éléments à l'intérieur des activités emboités et fournit le contexte dans lequel ils sont exécutés. Le contexte le plus à l'extérieur est la définition même du processus.

Deux constructions en langage BPEL requièrent l'usage des *scopes.* La première activité dans le gestionnaire d'événement *onEvent* et dans l'activité *forEach* doit être un *scope.* Ces *scopes* possèdent la définition de la variable du gestionnaire d'événement et la variable compteur de boucles (l'attribut *counterName* du *foreEach),* respectivement. Dans le cas du gestionnaire d'événement, le *scope* emboité peut aussi contenir la définition du lien de partenaire, du message échangé ou celle de l'ensemble corrélé utilisé par le gestionnaire d'événement. Si l'activité *invoke* contient la définition du gestionnaire

d'erreur ou celle du gestionnaire de compensation alors, elle est équivalente à un *scope* implicite emboité immédiatement dans l'activité *invoke.* Cette activité implicite suppose que le *name* de *invoke* comprend son attribut *suppressJoinFailure* et ses éléments : *sources* et *targets.*

1. **Cycle de vie d'un scope**

Un *scope* peut être utilisé comme une activité régulière (par exemple comme un élément fils d'une boucle loop). Le cycle de vie d'un *scope* commence avec la séquence d'initialisation pour les entités définies localement à l'intérieur du *scope*:

- Initialisation des variables et des liens de partenaire
- Instantiation des ensembles corrélés
- Installation des gestionnaires d'exceptions, de terminaison et d'événement.

Ces étapes sont exécutés indivisiblement (soit ils réussissent leur exécution soit l'erreur *bpel:scopeInitializationFailure* est lancée vers le *scope* parent). Après l'initialisation du *scope,* sa première activité est exécutée et tous les gestionnaires d'événements sont activés en parallèle, excepte l'activité initiale du *scope* qui est exécutée avant l'activation des gestionnaires d'événement.

Un *scope* finit son travaille avec ou sans succès. Il existe trois cas différents :

- Une terminaison normale: si la première activité se termine sans fautes et s'il n'existe pas d'activités orphelines de type message alors tous les gestionnaires d'événements sont désactivés (l'exécution de l'instance du gestionnaire d'événement est exigée pour finir), le gestionnaire de compensation est installé. Le *scope* finit avec succès.
- Une erreur interne : si l'erreur est lancée à l'intérieur du *scope* alors toutes les autres activités en court d'exécution et les instances du gestionnaire d'événement à l'intérieur du *scope* sont terminés et un gestionnaire d'erreur similaire est exécuté. Le *scope* finit sans succès.
- Terminaison externe : si un *scope* en court d'exécution reçoit un signal de terminaison(à cause d'une erreur externe ou une condition d'achèvement) alors toutes les autres activités en court d'exécution et les instances du gestionnaire d'événement à l'intérieur du *scope* sont terminés. Le scope finit sans succès.

2. Gestion d'erreur d'un scope

En BPEL, les gestionnaires d'erreur peuvent également être associes avec un scope pour traiter des situations d'exception locales (voir la figure 1.16).

```
 <scope>
<faultHandlers>
   <catch faultName="xyz:anExpectedError">...</catch>
   <catchAll><!-- deal with other errors -->
   …
   </catchAll>
</faultHandlers>
<sequence>
   <!-- do work -- >
</sequence>
</scope>
```

Figure 1.16 Gestion d'erreur d'un scope

Quand une erreur se produit à l'intérieur d'un scope alors ce dernier se termine sans succès. Avant la fin du traitement du scope, un gestionnaire d'erreur local peut traiter cette erreur. Les gestionnaires d'erreur peuvent eux même lancer de nouvelles erreurs ou relancer l'erreur capturée vers le scope emboité suivant ou vers le processus s'il n'y a plus de scope emboité.

3. Terminaison d'un travail en court d'exécution

Les scopes eux même peuvent influencer leur comportement de terminaison. Des cas typiques d'usage incluent l'exécution d'un travail propre ou envoyer un message à un partenaire métier. Après terminaison de la première activité du scope et toutes les instances du gestionnaire d'événements, le *terminationHandler* est exécuté (voir la figure 1.17).

```
 <scope>
<terminationHandler>
   <!-- clean up resources in case of forced termination -->
</terminationHandler>
<sequence>
   <!-- do work -->
</sequence>
</scope>
```

Figure 1.17. Gestion des terminaisons

Les gestionnaires personnalisés de terminaison peuvent contenir des activités BPEL qui incluent *compensate* et *compensateScope*. Ils ne peuvent pas propager les

erreurs au-delà de leur logique métier interne parce que la terminaison peut soit être causée par une autre erreur soit par une condition de terminaison d'une activité forEach.

4. Annulation d'un travail terminé

Le processus métier représente typiquement la longue exécution d'un travail qui ne peut être complété à l'intérieur d'une unique transaction atomique. Déjà les transactions ACID engagé pour cet effet créent des effets persistants avant que le processus termine. Des étapes appliquées spécifiquement à une compensation annulent ces effets si nécessaire. Dans un scope, la construction du langage pour renverser à l'avance les étapes d'un processus terminé est le *compensationHandler*. Il peut être invoqué après le succès de la terminaison de son scope associé, en utilisant l'activité *compensate* ou *compensateScope* (voir la figure 1.18).

```
 <scope name="S1">
<faultHandlers>
   <catchAll>
<compensateScope target="S2" />
   </catchAll>
</faultHandlers>
<sequence>
   <scope name="S2">
<compensationHandler>
<!-- undo work -- >
</compensationHandler>
<!-- do some work -->
   </scope>
   <!-- do more work -- >
   <!-- a fault is thrown here; results of S2 must be undone -- >
</sequence>
</scope>
```

Figure 1.18. Gestion d'une compensation

Le gestionnaire de compensation d'un scope est visible par rapport à l'état courant d'une instance du processus. L'état d'un scope terminé avec succès est sauvegardé pour permettre au gestionnaire de compensation de « continuer » a travaillé sur lui plus tard. L'état des scopes internes est partagé avec d'autres travaux concurrents tels que toute concurrence et considération isolée appliquées au gestionnaire de compensation dans le même sens comme pour l'activité primaire du scope.

Les gestionnaire de compensation sont invoqués par l'activité *compensate* ou *compensateScope*. Lesquelles peuvent résider dans un gestionnaire d'erreur *(fault handler),*un gestionnaire de compensation *(compensate handler*) ou une gestionnaire de terminaison *(termination handler)* (référencés par FCT-handlers), d'un scope interne immédiat. Une activité *compensate* pousse le gestionnaire de compensation de tous les

scopes fils terminés avec succès ou pas encore compensés, à s'exécuter dans un ordre par défaut. Une activité *compensateScope* provoque l'exécution du gestionnaire de compensation de tous les scopes fils terminés avec succès et pas encore compensée. Si une compensation d'un gestionnaire associé à un scope est contenue dans une structure répétitive ou gestionnaire d'événement alors de multiples instances du gestionnaire de compensation existent. Chacune associée avec une instance de scope correspondante. Finalement, chaque instance du gestionnaire de compensation peut en retour provoquée l'invocation des gestionnaires de compensation des scopes internes. L'ensemble de tous les instances des gestionnaires de compensation invoqués par l'activité *compensate* ou *compensateScope* est appelé « groupe d'instance de gestionnaires de compensation » *compensation handler instance group.*

Si une erreur est lancée dans un gestionnaire de compensation alors l'erreur est propagée au scope contenant l'activité *compensate* ou *compensatScope* qui invoque le groupe d'instance des gestionnaires de compensation. Avant le commencement du gestionnaire des erreurs correspondant, la terminaison de tous les travaux en court d'exécution comprend tous les gestionnaires de compensation du groupe encore en court d'exécution.

5. Gestion d'événements

Chaque scope aussi bien le processus lui-même peut définir les gestionnaires d'événements, ils sont utilisés pour traiter les messages requêtes des services web arrivants en parallèle sur l'activité première du scope du processus.

VI. Conclusion

Au niveau recherche, il existe des langages d'orchestration et des implémentations commerciales de ces langages permettant d'exécuter des processus bases sur le concept d'orchestration. BPEL est une spécification qui représente un langage orienté à la modélisation des processus qui utilisent les services web pour leur implémentation.
Néanmoins, BPEL est un langage de bas niveau, proche des langages de programmation ce qui le rend difficile à comprendre.

Chapitre 2 : La Logique de Réécriture et le Système Maude

Introduction

La réécriture est un paradigme général d'expression de calcul dans différentes logiques computationnelles. Les calculs prennent la forme de règles dans une syntaxe donnée. Dans une logique équationnelle, les calculs résultent de l'interprétation d'équations entre termes. Dans une logique de satisfaction de contraintes, une règle de réécriture peut être interprétée de deux manières, soit comme une transformation syntaxique, soit comme une inférence logique d'une nouvelle formule [6]. Par contre, dans la logique de réécriture, réécrire un terme consiste à le remplacer par un terme équivalent, en vertu des lois de l'algèbre des termes [7].L'idée essentielle de la logique de réécriture [7] est que la sémantique de la réécriture peut être rigoureusement changée d'une manière très fructueuse. Il a été largement démontré qu'elle permet de raisonner parfaitement sur le comportement des systèmes concurrents. Donc la logique de réécriture est un modèle de calcul et un cadre sémantique expressif pour la concurrence, le parallélisme, la communication, et l'interaction.

Dans ce chapitre, nous présentons les concepts de base de la logique de réécriture et du système Maude.

I. La logique de réécriture

La logique de réécriture est une logique de changement concurrent qui peut traiter l'état et le calcul des systèmes concurrents. Elle a été introduite par José Meseguer [8] comme une conséquence des travaux sur les logiques générales. Dès lors, cette logique a été largement utilisée pour spécifier et analyser des systèmes et langages dans différents domaines d'applications. Donc la logique de réécriture offre un cadre formel nécessaire pour la spécification et l'étude du comportement des systèmes concurrents. En effet, elle permet de raisonner sur des changements complexes possibles correspondant aux actions atomiques axiomatisées par les règles de réécriture. Le point clé de cette logique est que la *déduction* logique, qui est intrinsèquement concurrente, correspond au calcul dans un système concurrent [9, 10].

Un autre aspect important de la logique de réécriture est qu'elle représente un cadre logique et sémantique général dans lequel des langages et des modèles de calcul de nature largement différentes ont été représentés. Dans ce contexte, nous pouvons citer sans être exhaustifs, les systèmes de transitions étiquetés [11], les réseaux de Petri [12].. etc. En plus, cette logique peut constituer une base formelle rigoureuse pour la description des architectures logicielles [13, 14, 15].

1. Théorie de réécriture

Dans la logique de réécriture, un système concurrent est décrit par une théorie de réécriture $R = (\Sigma, E, L, R)$ où (Σ,E) désigne la signature (une théorie équationnelle) définissant la structure algébrique particulière des états du système (multi-ensemble, arbre binaire…) qui sont distribués selon cette même structure. La structure dynamique du système est décrite par les règles de réécriture étiquetées R (L est un ensemble d'étiquettes de ces règles). Les règles de réécriture précisent quelles sont les transitions élémentaires et locales possibles dans l'état actuel du système concurrent.

Chaque règle (notée $[t] \rightarrow [t']$) correspond à une action pouvant survenir en occurrence avec d'autre actions. Donc la logique de réécriture est une logique qui capture clairement le changement concurrent dans un système.

Définition (Théorie de réécriture étiquetée)

Une théorie de réécriture R est un 4-uplet (Σ, E, L, R) tel que :

1. *Σ est un ensemble de symboles de fonctions, et de sortes.*
2. *E un ensemble de Σ-équations (l'ensemble des équations entre les Σ-termes).*
3. *L est un ensemble d'étiquettes.*
4. *R est un ensemble de règles de réécriture, défini ainsi* $R \subseteq L \times (T_{\Sigma,E}(X))^2$ *;*

chaque règle est un couple d'éléments, le premier est une étiquette, le second est une paire de classes d'équivalence de termes $T_{\Sigma,E}(X)$ *sur la signature (Σ,E), modulo les équations E, avec* $X = \{x_1, \ldots, x_n, \ldots\}$ *un ensemble infini et dénombrable de variables.*

La réécriture opère sur les classes d'équivalence de termes, modulo l'ensemble des équations *E*. Ainsi, la réécriture est libérée des contraintes syntaxiques de la représentation des termes pour bénéficier d'une grande flexibilité dans le choix des structures de données. Pour une règle de réécriture de la forme $r([t] , [t'] , C_1, \ldots, C_k)$, la notation suivante est utilisée,

$$r : [t] \rightarrow [t'] \ \textbf{if} \ C_1 \wedge \ldots \wedge C_k,$$

où une règle ***r*** exprime que la classe d'équivalence contenant le terme ***t*** peut se réécrire en la classe d'équivalence contenant le terme ***t'*** si la condition de la règle $\boldsymbol{C_1 \wedge \ldots \wedge C_k}$ est vérifiée. Cette dernière est appelée condition de la règle et peut être abrégée par la lettre ***C***, et la règle de réécriture, dans ce cas, est dite conditionnelle. La partie conditionnelle d'une règle peut être vide, dans ce cas les règles sont appelées règles de réécriture inconditionnelles et sont notés par

$$r : [t] \rightarrow [t']$$

Une règle de réécriture peut être paramétrée par un ensemble de variables ***{x₁, … , xₙ}***

qui apparaissent soit dans ***t, t'*** ou ***C***, et nous écrivons :

$$r : [t(x_1, \dots, x_n)] \rightarrow [t'(x_1, \dots, x_n)] \ \textbf{if} \ C(x_1, \dots, x_n)$$

2. Déduction dans la logique de réécriture

Le calcul dans un système concurrent est une séquence de transitions(règles de réécriture) exécutées à partir d'un état initial donné. Il correspond à une preuve ou une déduction dans la logique de réécriture.

Cette déduction est intrinsèquement concurrente et permet de raisonner correctement sur l'évolution du système d'un état à un autre [16, 8].

Définition (Principe de déduction). *Etant donné une théorie de réécriture* R*,= (Σ, E, L, R), nous disons que la séquence* $[t] \rightarrow [t']$ *est prouvable dans* R *et on écrit* $R \vdash [t] \rightarrow [t']$ *si et seulement si* $[t] \rightarrow [t']$ *est obtenue par une application finie des règles de déduction suivantes:*

1. ***La réflexivité*** *: pour chaque terme* $[t] \in T_{\Sigma,E}(X)$, $\overline{[t] \rightarrow [t]}$ *où* $T_{\Sigma,E}(X)$ *est l'ensemble des Σ-termes avec variables construits sur la signature* Σ *et les équations E.*

2. ***La congruence*** *: pour chaque fonction* $f \in \sum_n$, $n \in N$,

$$\frac{[t_1] \rightarrow [t_1'] \dots [t_n] \rightarrow [t_n']}{[f(t_1, \dots, t_n)] \rightarrow [f(t_1', \dots, t_n')]}$$

3. ***le remplacement*** *: pour chaque règle* $r : [t(x_1, \dots, x_n)] \rightarrow [t'(x_1, \dots, x_n)]$ *dans R :*

$$\frac{[w_1] \rightarrow [w_1'] \dots [w_n] \rightarrow [w_n']}{[t(\overline{w} / \overline{x})] \rightarrow [t'(\overline{w}' / \overline{x})]}$$

Sachant que $t(\overline{w}/\overline{x})$ *dénote la substitution simultanée de* x_i *par* w_i *dans* t *avec* $\overline{x}$ *représentant* $x_1, \ldots, x_n$.

4. ***la transitivité*** *:* $\dfrac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

De manière générale, la déduction dans la logique de réécriture est une itération des étapes suivantes :

1. La règle de *remplacement* identifie toutes les règles de réécriture dont le membre gauche correspond à un sous-terme de l'état global courant. Comme la logique de réécriture est une logique de changement, la règle de réflexivité, appliquées aux sous-termes non identifiés, les transforme mais en eux-mêmes.
2. Les règles de réécriture, identifiées par la règle de *remplacement* ainsi que la règle de *réflexivité*, sont exécutées en concurrence et indépendamment les unes des autres. La règle de *congruence* compose les effets, membres droits, de ces règles pour construire le nouveau terme global.

Les étapes (1) et (2) sont réitérées jusqu'à ce qu'il n'y ait plus de règle applicable.

3. Enfin, la règle de *transitivité* construit la séquence des réécritures faites du terme initial jusqu'au terme final. La séquence ainsi construite correspond à un calcul possible dans le système concurrent.

Remarque : Les règles de *congruence* et de *remplacement* peuvent être vues sous un autre angle. La première règle exprime que des règles de réécriture disjointes (des règles de réécriture sont disjointes si elles n'ont pas de sous-termes communs) peuvent être exécutées en concurrences. Alors que la seconde règle permet des réécritures imbriquées, i.e., imbrication de la réécriture des sous termes $w_i \rightarrow w'_i$ dans celle du terme composite $t \rightarrow t'$. Elle indique que deux sous termes différents peuvent être réécrits en parallèle même si leurs racines, terme composite, ne sont pas disjointes [17].

La *réflexivité* introduit le calcul « identité », c'est-à-dire que les sous-termes qui ne sont pas susceptibles de changer suite à l'application d'une étape de réécriture doivent être réécrits en utilisant la règle de *réflexivité*. Cette règle découle du fait que la logique de réécriture est une logique de « changement », tous les termes et sous- termes

doivent être changés (substitués) durant une étape de réécriture. Les termes qui ne changent pas se réécrivent donc en eux même.

Réflexivité

Congruence

Remplacement

Transitivité

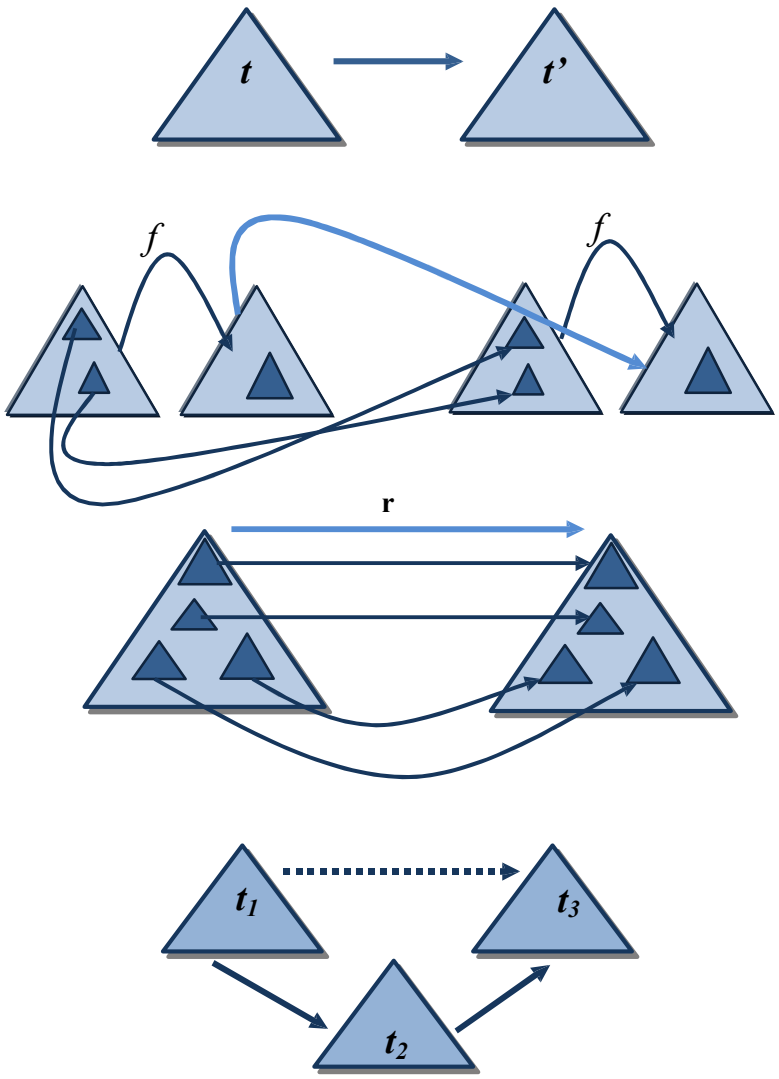

Figure 2.1 - Représentation graphique des règles de déduction

La *transitivité* détermine la composition séquentielle des étapes de réécriture. Cette règle offre la possibilité de construire la séquence de déduction à partir du terme initial jusqu'au terme final (à partir du quel plus aucune réécriture n'est possible).

La *congruence* spécifie que les réécritures peuvent être emboitées dans des contextes plus larges. Autrement dit, la composition parallèle d'un ensemble de transformations locales produit un état composite et cohérent et qui sera le nouveau terme global.

La règle de déduction qui semble la plus complexe est le *remplacement* qui identifie les termes à remplacer dans une expression d'un terme global représentant.

L'état actuel du système concurrent en faisant les substitutions de variables appropriées.

3. Réécriture concurrente

Le calcul dans un système concurrent (représenté par une séquence d'étapes de réécritures concurrentes exécutées à partir d'un état initial donné) correspond à une preuve ou une déduction dans la logique de réécriture en utilisant les règles de déduction appropriées (*réflexivité*, *congruence*, *remplacement*, *transitivité*). Par conséquence, une théorie de réécriture est vue comme une spécification exécutable du système qu'elle formalise. Les règles de réécriture *R* de cette théorie décrivent les transitions élémentaires possibles dans un système. Les règles de déduction de la logique de réécriture permettent de raisonner correctement sur les transitions générales possibles dans un système.

Définition (réécriture concurrente) *Etant donné une théorie* $\mathfrak{R}$ ***=(Σ,E,L,R),*** *une* ***(Σ,E)-formule [t] → [t']*** *est dite:*

1. $\mathfrak{R}$ ***-réécriture concurrente 0-step*** *si et seulement si elle est dérivée à partir de* $\mathfrak{R}$ *par une application finie des règles de déduction* ***1*** *et* ***2*** *(dans ce cas les termes [t] et [t'] coïncident nécessairement) ;*
2. $\mathfrak{R}$ ***-réécriture concurrente 1-step*** *si et seulement si elle est dérivée à partir de* $\mathfrak{R}$ *par une application finie des règles de déduction* ***1, 2*** *et* ***3,*** *avec au moins une application de la règle* ***3*** *(règle de remplacement). Si toutefois la règle* ***3*** *est*

appliquée une seule fois seulement, on appelle la formule une étape de $\mathfrak{R}$ –

réécriture *séquentielle ;*

3. $\mathfrak{R}$***-réécriture concurrente*** *(ou juste une* ***réécriture****) si et seulement si elle est dérivée à partir de* $\mathfrak{R}$ *par une application finie des règles de déduction* ***1, 2, 3*** *et* ***4.***

II. Extension de la logique de réécriture

La logique de réécriture dépend essentiellement de la logique équationnelle sous- jacente. Les auteurs de [18] précisent que la généralisation vers des logiques équationnelles plus expressives implique des versions plus expressives de la logique de réécriture en permettant notamment des conditions plus générales dans la partie conditionnelle des règles de réécriture et en introduisant formellement la notion d'interdiction de la réécriture de sous-termes relatifs à certaines positions dans des opérateurs. Ainsi, ces extensions de la logique de réécriture, proposées par Bruni et Meseguer, développent de nouvelles bases sémantiques pour une version révisée de cette logique qui supporte plusieurs caractéristiques nouvelles dont l'expressivité a été trouvée très utile dans la pratique. La révision de l'expression de son formalisme a été proposée selon plusieurs dimensions. D'abord en choisissant la logique équationnelle d'appartenance (membership) comme logique équationnelle fondamentale. Ensuite en permettant des conditions très générales dans les règles conditionnelles de réécriture. La troisième dimension permet de déclarer certains arguments d'opérateurs comme gelés (frozen), pour les bloquer en réécriture.

III. Réflectivité et stratégie de réécriture

La logique de réécriture est réflective dans un sens mathématique précis, c'est-à-dire, il existe une théorie finie de réécriture U qui est universelle dans le sens où on peut représenter dans cette théorie U (comme des termes) toute autre théorie de réécriture finie R (y compris la théorie U elle-même) [17]. Ainsi, il est possible de simuler dans la théorie U toute réécriture inférée (déduite) dans la théorie de réécriture R. Par conséquent, il existe une représentation finie $\overline{R}$ sous forme de termes de U de toute théorie de réécriture R, une représentation $\overline{t}$ et $\overline{t'}$ sous forme de termes de U de tous termes t, t' et R ,et une représentation $<\overline{R} ; \overline{t}>$ de tous couple (R, t) vérifiant $R : t \rightarrow t' \Leftrightarrow (U : <\overline{R} ; \overline{t}> \rightarrow <\overline{R} ; \overline{t'}>)$

La réflexion permet de guider le processus de déduction induit par une théorie de réécriture R au niveau objet par le biais de stratégies de réécriture dont la sémantique peut être définie à l'intérieur de la logique de réécriture par des théories de réécriture définies à un méta-niveau. Dans ce cas, de telles stratégies de réécriture constituent des procédures d'inférence particulières spécifiques à la théorie R. Ainsi, la propriété de réflexion permet à la logique de réécriture de décrire fidèlement le comportement de certains systèmes dont la sémantique complète ne peut être spécifiée simplement par un ensemble de règles de réécriture mais nécessite aussi la spécification de procédures d'exécution particulières de ces règles (par exemple, en précisant un ordre d'exécution spécifique).

IV. Système MAUDE

Maude est un langage déclaratif qui implémente correctement tous les concepts théoriques de la logique de réécriture. Il constitue un système de haute performance, défini par J. Meseguer [19], supportant à la fois la spécification exécutable et la programmation déclarative des logiques équationnelles et de réécriture pour un grand nombre d'applications. En général, un programme écrit dans le langage déclaratif Maude représente une théorie de réécriture, c'est-à-dire, une signature et un ensemble de règles de réécriture. Le calcul dans ce langage correspond à la déduction en logique de réécriture en utilisant les axiomes spécifiés dans ces théories/programmes.

Les unités basiques de spécification ou de programmation dans Maude sont appelées des *modules*. Par conséquent, on différencie trois types de modules : les modules fonctionnels pour implémenter les théories équationnelles. Les modules système implémentent les théories de réécriture et définissent le comportement dynamique d'un système. Les modules orientés-objet implémentent les théories de réécriture orientées objet (ils peuvent être réduits à des modules systèmes).

1. Modules Fonctionnels

Un module fonctionnel est introduit par les mots clés *fmod <Corps du module> endfm* où le corps du module spécifie une théorie (Σ, EUA, Φ) dans la logique équationnelle d'appartenance. La signature Σ inclut des sortes (indiqués par le mot clé sort), des sous-sortes (spécifiés par le mot clé *subsort*) et des opérateurs (introduits avec le mot clé *op*).

La syntaxe des opérateurs est définie par les utilisateurs en indiquant la position des arguments par le symbole (_). Certains de ces arguments peuvent être spécifiés comme figés en

utilisant le mot clé frozen (*PositionArgument*). L'ensemble ***E*** désigne les équations et les tests d'appartenance (qui peuvent être conditionnels) et **A** est un ensemble d'axiomes équationnels introduits comme attributs de certains opérateurs dans la signature **Σ** tels que les axiomes d'associativité (spécifiée par le mot clé *assoc*), de commutativité (spécifiée par le mot clé *comm*) ou d'identité (spécifiée par le mot clé id:). Ces derniers sont définis de manière à ce que les déductions équationnelles se fassent modulo les axiomes de *A*. Les équations sont spécifiées par le mot clé *eq* ou le mot clé ***ceq*** (pour les équations conditionnelles) et les tests d'adhésion ou d'appartenance sont introduits avec les mots clés *mb* ou *cmb* (pour les tests d'appartenance conditionnels). Une condition liée à une équation ou à un test d'appartenance peut être formée par une conjonction d'équations et de tests d'adhésions inconditionnels.

Dans un module fonctionnel, les équations sont utilisées comme des règles de simplification par lesquelles chaque expression, après substitution des variables, peut être évaluée et simplifiée à sa forme réduite dite *forme canonique*. Le résultat de la simplification d'un terme initial est unique quel que soit l'ordre d'application des équations. Les variables peuvent être déclarées dans les modules avec les mots clés var ou vars, ou introduites directement dans les équations et les tests d'adhésion, sous la forme d'une expression var : sort.

2. Modules systèmes

Les modules systèmes sont utilisés pour définir le comportement dynamique des systèmes concurrents en enrichissant les modules fonctionnels par un ensemble de règles de réécriture. Ils sont introduits par les mots clés *mod<Corps du module>*
endm où le corps du module spécifie une *théorie de réécriture* $\mathfrak{R} = (\Sigma, EUA, \Phi, R)$

avec (Σ, EUA, Φ) la théorie équationnelle sous-jacente. Les règles de réécriture R sont

introduites avec les mots clés *rl* ou *crl*. Elles sont spécifiées dans Maude avec la

syntaxe : crl [*l*] : *t* => *t'* **if** *cond* .

Si la règle est non conditionnelle, le mot clé *crl* est remplacé par *rl* et la clause

« **if** *cond* » est omise.

3. Modules orientés objet

Il est possible de présenter un système d'objets concurrents comme une théorie de réécriture avec une syntaxe plus appropriée pour décrire les concepts de base du paradigme objet. Les modules orientés objet sont supportés par le système Full- Maude [17]. Notons que Full Maude est une extension de Maude (Core Maude) dont le code est écrit en Maude, ce qui enrichit le langage Maude avec un module algébrique très puissant et extensible. Ces modules orientés objet sont introduits par les mots clés : (*omod* < *corps du module*> *endom*) où le corps du module est une *théorie de réécriture* $\mathfrak{R} = (\Sigma, EUA, \Phi, R)$. Ils supportent la spécification et la manipulation des objets, des messages, des classes et de l'héritage.

Un système orienté objet concurrent dans ce cas est modélisé par un multi-ensemble d'objets et de messages juxtaposés, où les interactions concurrentes entre les objets sont régies par des règles de réécriture. Un objet est représenté par le terme $< O : C \mid a_1 : v_1 , \ldots , a_n : v_n >$, où O est le nom de l'objet instance de la classe C, $a_i, \in 1..n$, les noms des attributs de l'objet, et v_i, leurs valeurs respectives.

La déclaration des classes suit la syntaxe : *class* $C \mid a_1 : s_1 , \ldots , a_n : s_n .$, où C est le nom de la classe et s_i est la sorte de l'attribut a_i. Il est aussi possible de déclarer des sous classes et bénéficier ainsi de la notion d'héritage. Les messages sont déclarés en utilisant le mot clé *msg*. La forme générale d'une règle de réécriture dans la syntaxe orientée objet de Maude est :

crl [r] : $M_1 \ldots M_n < O_1 : F_1 \mid a_1 > \ldots < O_m : F_m \mid a_m > \Rightarrow < O_{i1} : F'_{i1} \mid a'_{i1} > \ldots$

$< O_{ik} : F'_{ik} \mid a'_{ik} > M_1' \ldots M_p'$ ***if*** *Cond* .

Où r est l'étiquette de la règle, $M_s, s \in 1..n,$ et $M'_u, u \in 1..p$ sont des messages, $O_i, i \in 1..m,$ et $O_{il}, l \in 1..k$, sont des objets, et *Cond* est la condition de la règle. Si la règle est non conditionnelle, nous remplaçons le mot clé *crl* par *rl* et nous enlevons la clause *if Cond*.

Notons que le système Full-Maude offre un support additionnel pour la programmation orientée objet avec les notions de classes, sous-classes et une syntaxe plus conviviale des règles de réécriture. Ainsi, il permet au système Maude de supporter la modélisation orientés objet en fournissant le module prédéfini *CONFIGURATION*. Dans ce module, les sortes représentant les concepts essentiels des objets, classes, messages et configurations, sont déclarés.

4. Modules prédéfinis

Les modules prédéfinis de Maude sont stockés dans une librairie spécifique et peuvent être importés par d'autres modules définis par l'utilisateur. Ils sont introduits dans les fichiers sources de Maude *prelude.maude* et *model-checker.maude* : comme par exemple les BOOL, STRING et NAT. Ces modules déclarent les sortes et les opérations pour manipuler, respectivement, les valeurs *booléennes*, les *chaînes de caractères* et les *nombres naturels*. Le fichier *model-checker.maude* contient les modules prédéfinis interprétant les outils nécessaires pour l'utilisation du *LTL Model Checker* de Maude.

V. Exécution et analyse formelle sous Maude

Dans un module écrit en Maude, les règles de réécriture constituent les unités élémentaires d'exécution. Elles interprètent les actions locales du système modélisé, et peuvent être exécutées dans un temps constant et de manière concurrente (à n'importe quel moment). Maude nous offre la possibilité de simuler l'exécution de telles réécritures (via des règles de réécriture) ou des réécritures équationnelles (via des équations) dans un module M par l'implémentation des deux commandes : *reduce* et *rewrite*.

La commande *reduce* (abrégée par *red*) permet à un terme initial d'être réduit par application des équations et des axiomes d'adhésion dans un module donné. Elle se présente sous la syntaxe suivante :

Reduce {in module :} term .

La commande de réécriture *rewrite* (abrégée par *rew*) et la commande de réécriture équitable *frewrite*(abrégée par *frew*) exécutent une seule séquence de réécriture (parmi plusieurs séquences possibles) à partir d'un terme initialement donné suivant la syntaxe :

rewrite {in module : } term .

frewrite {in module : } term .

Ces commandes permettent de réécrire un terme initial en utilisant les règles, les équations et les axiomes d'adhésion dans le module spécifié.

1. Analyse formelle et vérification des propriétés

La vérification formelle de modèles sur les systèmes spécifiés en Maude s'effectue à l'aide d'outils disponibles autour du système Maude. Parmi ces outils, on distingue un outil d'analyse d'accessibilité (la commande *search*) et un module de vérification par *model checking* de propriétés exprimées en logique linéaire temporelle (LTL) [17]. Pour analyser toutes les séquences de réécritures possibles à partir d'un état (terme) initial t_0, on utilise la commande

search. Celle-ci recherche si des états correspondants à des patterns donnés et satisfaisant certaines conditions, peuvent être accessibles à partir de t_0. L'exécution de cette commande effectue un parcours en profondeur de l'arbre de calcul (arbre d'accessibilité), généré lors de cette recherche, afin de détecter les violations d'invariants dans les systèmes à états infinis [17].

L'analyse formelle des systèmes par *Model Checking* [17] est un ensemble de techniques pour vérifier automatiquement des propriétés temporelles relatives au comportement des systèmes. Le *Model Checking* permet de vérifier la satisfiabilité d'une propriété donnée dans un état ou un ensemble d'états, en faisant une exploration exhaustive de l'ensemble des états accessibles à partir d'un état initial. Il reçoit en entrée une abstraction du comportement du système (un système de transitions), représentée par une structure Kripke K, et une propriété ϕ de ce système,formulée dans une certaine logique temporelle, et répond si l'abstraction satisfait ou non la formule ϕ c'est-à-dire, si K |= ϕ. L'intérêt du *Model Checking* est qu'il retourne une trace d'exécution du système violant la propriété lorsque cette dernière est non valide.

Logique temporelle linéaire (LTL)

La logique temporelle linéaire (LTL) est une extension de la logique classique avec des opérateurs temporels, tels que G et F (représentant respectivement « globalement », « finalement ou fatalement ») qui permettent d'exprimer des propriétés portant sur l'exécution d'une séquence d'états. Elle est dite temporelle car elle décrit le séquencement d'évènements observés dans un système. Cette logique permet de spécifier des propriétés intéressantes pour les systèmes concurrents notamment les propriétés de ***sûreté***, d'***accessibilité***, de ***vivacité*** et d'***équité***.

- ***Sûreté*** : quelque chose de mauvais n'arrive jamais.
- ***Accessibilité*** : une certaine situation peut être atteinte.
- ***Vivacité*** : quelque chose de bon est toujours possible.
- ***Equité*** : quelque chose se répètera infiniment souvent.

Les formules LTL sont construites à partir de variables propositionnelles appelées propositions atomiques, d'opérateurs booléens (¬, ∧, ∨, ⇒, ⇔), et d'opérateurs temporels, **F** (Futur), **G** (Global), **U** (Until), **X** (neXt). Les propriétés atomiques permettent de décrire les états du système : un état ***t*** est dit étiqueté par une

proposition atomique ϕ si ϕ est vraie dans ***t***. Les opérateurs temporels permettent de relier des états du système au sein d'une séquence d'exécution.

• La formule **X** *f* indique que la formule LTL *f* doit être vérifiée à l'instant suivant immédiat le long de l'exécution (*neXt*).

• La formule *f* **U** *g* indique que *f* est toujours vrai jusqu'à un état où *g* est vrai (*Until*).

• La formule G *f* signifie *f* est toujours vérifiée (*Generally*) dans toute l'exécution.

• La formule **F** *f* indique que *f* doit être vérifiée plus tard au moins dans un état de l'exécution (*Fatalement*).

2. Le Model Checker LTL de Maude

L'outil de model checking qu'offre Maude fait appel à la logique LTL. Il exige que le système à vérifier, décrit par une *théorie de réécriture* $\mathfrak{R} = (\Sigma, E, \Phi, R)$, soit à état fini, c'est-à-dire qu'à partir d'un état initial [t0], l'ensemble des états accessibles défini par :
$\{[u] \in T_{\Sigma,E} \mid \mathfrak{R} \vdash [t_0] \rightarrow [u]\}$ soit fini. Donc, le LTL model checker de Maude prend en entrée une théorie de réécriture $\mathfrak{R}$ dont il génère la structure Kripke K($\mathfrak{R}$, State) sous-jacente ainsi qu'une formule temporelle LTL ϕ et vérifie si cette structure K($\mathfrak{R}$, State) satisfait la formule ϕ. Si cette formule n'est pas valide, le LTL model checker retourne un contre-exemple qui montre une séquence d'états menant à la violation de cette formule. Cet outil est implémenté sous la forme d'un module, spécifié en termes de la logique de réécriture, dans le langage Maude. Ce dernier importe d'autres modules prédéfinis (spécifiés également dans le langage Maude) :

• Le module *LTL-SIMPLIFIER* implémente des procédures formelles pour simplifier la formule LTL exprimant une propriété.

• Le module *SATISFACTION* spécifie la syntaxe et la sémantique de l'opérateur de satisfaction (|=) indiquant si une formule donnée est vraie ou fausse dans un certain état.

• Le module *SAT-SOLVER* permet de vérifier la satisfiabilité et la topologie d'une formule spécifiée en logique LTL.

VI. Conclusion

Nous avons présenté, dans ce chapitre, les notions relatives à la compréhension des concepts de base de la logique de réécriture, du système Maude. Dans la section réservée à la logique de réécriture, nous nous sommes concentrés sur la présentation de son aspect théorique. Nous avons montré son pouvoir expressif pour décrire naturellement et d'une manière intuitive le comportement des systèmes concurrents. Cette description permet de révéler toutes les actions qui peuvent se produire en parallèle, décrivant ainsi correctement le comportement de ces systèmes selon une sémantique de vraie concurrence. Les déductions faites sur l'évolution des états d'un système représentent les calculs dans le système concurrent.

Nous avons présenté également, le système Maude. Il s'agit d'un système de haute performance et d'un langage de spécification formelle implémentant correctement tous les concepts théoriques de la logique de réécriture. Le système Maude supporte aussi une large gamme de techniques de vérification formelles, implémentés en langage Maude.

Chapitre 03 : K Framework

Introduction :

Présenté par Grigore Rosu en 2003 [22] pour enseigner les langages de programmation, développer et affiner depuis ce temps, le Framework K est utilisé pour définir des langages ainsi de spécifier des sémantiques de façon formelle et de pouvoir les manipuler. basé sur la réécriture qui apporte un ensemble de point forts des outils existants tel que l'environnement *Centaur* [41], en proposant un langage permettant de définir des sémantiques formelle à l'aide des règles d'inférence qui étaient traduite en prédicats Prolog, ainsi un environnement de débogage des règles de sémantique et l'outil *Coq* [40] ,qui définit la sémantique selon un style fonctionnel comme une fonction Coq.

Le Framework K a été utilisé pour formaliser plusieurs langages en développons des outils d'analyses et de vérification par exemple C [23], Python [24], Java [25], Scheme [26] ainsi que Haskell [27], JavaScript [28], LLVM[29]..et d'autres.

Le but du K Framework est de démontrer que la spécification formelle des langages de programmation peut être à la fois simple, analysable, expressive et évolutive.

L'un des questions à poser pourquoi définissant un langage en utilisant le Framework K ou autre outils alors que l'art et l'état actuel est d'implémenter des interpréteurs, des compilateurs et des outils d'analyse formelle pour chaque langage ; ce n'est pas seulement rentable parce que la plupart des outils ré-implémentent les mêmes techniques et les algorithmes mais aussi la tendance face aux erreurs.

Considérons l'exemple du programme C suivant :

```
int main(void) {
int x = 0;
return (x = 1) + (x = 2);
}
```

Quelle est l'évaluation de ce programme ?

- Selon le C standard le résultat est indéfinie ;
- Avec les compilateurs GCC4, MSVC la valeur retournée est 4.
- Avec les compilateurs GCC3, ICC, Clang la valeur retournée est 3.

Si une erreur sémantique est trouvée, elle doit être fixée dans chaque outil séparément ce qui est pénible et démotivant [23].

Dans ce chapitre, nous allons présenter le K Framework et les travaux de recherche en relation.

I. K Framework

Ecrire la première définition K

Dans cette section, nous allons guider le lecteur à travers le processus d'écriture d'une simple définition d'un langage à l'aide de l'outil K, en utilisant la définition du langage EXP présenté dans la figure 3.1.

1. Les ingrédients de base

Lorsque vous commencez à écrire des définitions à l'aide de l'outil K [30] , il est recommandé d'essayer la définition aussi souvent que possible, la capture et le traitement des problèmes durant le début de développement facilites la correction. Par conséquent, nous allons commencer par montrer comment obtenir une définition vérifiable le plus tôt possible dans le processus de développement.

1.1. Modules

L'outil K fournit des modules pour grouper les caractéristiques du langage [30]. On définit un module par la syntaxe suivante :

module<NAME>

…..

end module

```
1  module EXP-SYNTAX

3    //@ Arithmetics Syntax
4    syntax Exp ::= #Int
5                 | Exp "+" Exp [strict] //addition
6                 | Exp "*" Exp [strict] //multiplication
7                 | Exp "/" Exp [strict] //division
8                 | Exp "?" Exp ":" Exp [strict(1)]
9                 | Exp ";" Exp [seqstrict]

11   //@ Input / Output Syntax

13   syntax Exp ::= "read"
14                | "print" Exp [strict]


17   //@ Concurrency features
18   syntax Exp ::= "spawn" Exp
19                | "rendezvous" Exp [strict]
20 end module

22 module EXP
23   imports EXP-SYNTAX
24   syntax KResult ::= #Int
25   configuration
26     ⟨k color="green" multiplicity="*"⟩ $PGM:K ⟨/k⟩
27     ⟨streams⟩
28       ⟨in color="magenta" stream="stdin"⟩ .List ⟨/in⟩
29       ⟨out color="Fuchsia" stream="stdout"⟩ .List ⟨/out⟩
30     ⟨/streams⟩

32  //@ Arithmetics Semantics

34   rule I1:#Int + I2:#Int => I1 +Int I2
35   rule I1:#Int * I2:#Int => I1 *Int I2

37   rule I1:#Int / I2:#Int => I1 /Int I2
38                             when I2 =/=Bool 0
40   rule 0 ? _ : E:Exp => E
41   rule I:#Int ? E:Exp : _ => E when I =/=Bool 0

43   rule _:#Int ; I2:#Int => I2


46   //@ Input / Output Semantics


49   rule ⟨k⟩ read => I:#Int ~⟨/k⟩
50        ⟨in⟩ ListItem(I) => . ~⟨/in⟩


54   rule ⟨k⟩ print I:#Int => I ~⟨/k⟩
55        ⟨out⟩~ . => ListItem(I) ⟨/out⟩



60   //@ Concurrency Semantics


63   rule ⟨k⟩ spawn E => 0 ~⟨/k⟩
64        (. => ⟨k⟩ E ⟨/k⟩)



69   rule ⟨k⟩ rendezvous I => 0 ~⟨/k⟩
70        ⟨k⟩ rendezvous I => 0 ~⟨/k⟩


73 end module
```

Figure 3.1 Définition du langage EXP

Où <NAME> est le nom identifiant le module. Dans la figure 3.1 les lignes de 1 à 20 définie le module EXP-SYNTAX en utilise uniquement des lettres majuscules et des tirets ,la définition de K nécessite d'avoir au moins un module principal, mais l'outil K prévoit au moins deux par défaut: <NOM> et <NOM>-SYNTAX. La séparation de la syntaxe du reste de la définition minimise les ambiguïtés dans l'outil. Pour importer un module, il faut ajouter la directive **imports**<NOM> après l'en-tête du module par exemple, la ligne 23 de la figure 3.1.
Plusieurs modules peuvent être importés en utilisant la même directive d'importations en additionnant leurs noms avec le symbole "+".

1.2. Compiler une définition

Les définitions K sont généralement stockées dans des fichiers avec l'extension '.k '.Supposons le fichier exp.k contenant le texte suivant:

module EXP-SYNTAX
end module
module EXP
imports EXP-SYNTAX
end module

Pour compiler cette définition et vérifier sa validité, on peut exécuter la commande suivante:

$ **kompile** exp

Le fichier <name>.k qui sera compilé est supposé par défaut contient le module <NAME>.

1.3. L*e*s commentaires :

L'outil K introduit des commentaires similaire au langage C, "/ /" pour commenter une seul ligne et "/* ……*/" pour commenter plusieurs lignes plus des commentaires spécifique du *LaTeX* introduit par "//@" et "/*@".

2. Syntaxe du langage

Avant de donner une sémantique formelle à un langage, il faut d'abord fournir une syntaxe formelle, la syntaxe dans K est définit en utilisant la notation BNF, par exemple la déclaration sur les lignes de 4-9 de la figure 3.1, où les terminaux sont bornés entre guillemets et les non-terminaux commençant par une lettre majuscule (sauf le cas de built commence par " #")

```
4   syntax Exp ::= #Int
5                | Exp "+" Exp [strict] //addition
6                | Exp "*" Exp [strict] //multiplication
7                | Exp "/" Exp [strict] //division
8                | Exp "?" Exp ":" Exp [strict(1)]
9                | Exp ";" Exp [seqstrict]
```

3. La sémantique du langage :

Spécifier une sémantique en utilisant l'outil K se compose de trois parties [30] :
La fourniture d'une stratégie d'évaluation pour bien réorganiser les calculs, donnant la structure de la configuration pour maintenir l'état du programme et l'écriture des règles K pour décrire les transitions entre configurations.

3.1. les stratégies d'évaluation

Les stratégies d'évaluation servent de lien entre la syntaxe et la sémantique, en précisant dans quel ordre les arguments d'une structure du langage, doivent être évalués, par exemple les deux arguments de l'opérateur d'addition doivent être évaluées avant le calcul de leur somme, alors que pour l'opérateur conditionnel "_ ?_: _", seul le premier argument doit être évaluer.

L'outil K fournit deux attributs : **strict** et **seqstrict**. Chacun prend éventuellement une liste de numéros séparés par des espaces comme arguments, indiquant les positions sur lesquelles la construction doit **strict** (1 étant la position la plus à gauche),par exemple, la notation **strict** (1) dans la ligne 8 de la figure 3.1 indique que seul le premier argument de l'expression conditionnelle doit être évalué avant de donner la sémantique pour la construction elle-même.si aucun argument n'est fourni alors tous les postions sont considérées **strict**.

La seule différence entre **strict** et **seqstrict** est que ce dernier assure que les arguments sont évalués dans l'ordre donné comme un argument dans la liste, tandis que le premier permet le non-déterminisme.

L'outil K distingue une catégorie de termes, **KResult**, qui est utilisé pour déterminer quels sont les termes qui sont des valeurs ou des résultats. La déclaration **syntax** à la ligne 24 de la figure 3.1 spécifie que les valeurs sont des entiers dans le langage EXP.

3.2. Les Calculs (*Computation*)

La séquence d'évaluation est rendu possible en K par les structures de calcul (*computation structures*) appelées "*computation*", étendre la syntaxe abstraite d'un langage avec une structure de liste en utilisant le séparateur (↷ lire "Suivi par" ou "et ensuite" et écrit ~> en ASCII).

3.3. La Configuration

Dans K, l'état d'un programme en cours d'exécution est représenté par une configuration de structuré appelée cellules "*cells*" ,représentées à l'aide d'une notation de type XML.

les lignes 25 -30 de la figure 3.1 décrits à la fois la configuration initiale et la structure générale d'une configuration pour EXP, introduit par le mot-clé **configuration** et se compose d'une cellule étiqueté par k qui est destinée à maintenir le programme en cours d'exécution (notés ici par la variable $ PGM de Type K), et de cellules **streams** qui modélise les flux d'entrées/sorties d'un programme en exécution.

```
25  configuration
26    ⟨k color="green" multiplicity="*"⟩ $PGM:K ⟨/k⟩
27    ⟨streams⟩
28      ⟨in color="magenta" stream="stdin"⟩ .List ⟨/in⟩
29      ⟨out color="Fuchsia" stream="stdout"⟩ .List ⟨/out⟩
30    ⟨/streams⟩
```

Les cellules dans la déclaration de configuration peuvent contenir des attributs XML comme **color**,**stream** et **multiplicity** qui est utilisé pour spécifier le nombre de copies d'une cellule permise (0 ou 1 (?), 0 ou plus (*), ou d'un ou plusieurs (+)) variables dans la configuration initiale, par exemple, $ PGM: K dans la ligne 26 de la figure 3.1, Sont des espaces réservés. Ils sont initialisés au début de l'exécution.

Les seuls types cellules actuellement autorisés par l'outil K sont **List**, **Bag**, **Sets** et **Map**. Les injections correspondantes sont respectivement **ListItem**, **BagItem**, **SetItem**, et ↦ (Écrit en ASCII |->).

3.4. les règles (*Rules)*

Les règles K décrit comment un terme dans une configuration peut se changer en un autre terme, d'une manière similaire à celle d'une règle de réécriture: un terme du côté gauche d'une règle peut être remplacé par celui de côté droit. Dans l'outil K, les règles sémantiques sont introduites par le mot-clé **rule.**

Dans K, la réécriture est étendue en utilisant une idée appelée réécriture "*local rewriting*". En appuyant sur l'action de réécriture dans leurs contextes, ces règles peuvent omettre des parties d'un terme qui serait autrement dupliquées sur les deux côtés d'une règle de réécriture, cela est fait en permettant à plusieurs occurrences du symbole de réécriture ⇒ (écrit en ASCII =>),

reliant les parties des contextes correspondants qui sont changés par la règle (la gauche côté de ⇒) et leurs remplaçants correspondant (le côté droit de ⇒). Dans la notation *LaTeX*, ce remplacement est affiché sur l'axe vertical, à l'aide d'une ligne horizontale à la place du symbole de réécriture. Par exemple, la règle d'impression (lignes 54 et 55 de la figure 3.1) effectue deux changements dans la configuration: (1) l'expression d'impression est remplacée par son argument entier, (2) un élément de liste contenant cet entier remplace la liste vide dans la cellule de sortie (à savoir, il est ajouté à la liste)

```
54  rule ⟨k⟩ print I:#Int => I ~⟨/k⟩
55       ⟨out⟩~ . => ListItem(I) ⟨/out⟩
```

4. L'exécution de programme avec krun

Une fois la définition du langage est faite et compilé avec succès avec la commande **kompile**, la commande **krun** est utilisée pour exécuter le programme écrit dans le langage définie. Par défaut, **krun** a les mêmes restrictions que celles de la commande **kast** qui est utilisée pour voir l'arbre syntaxique abstrait.

```
$ cat p0.exp
3 * (4 + 6) / 2

$ krun p0.exp

⟨k⟩
  15
⟨/k⟩
⟨streams⟩
  ⟨in⟩
    .
  ⟨/in⟩
  ⟨out⟩
    .
  ⟨/out⟩
⟨/streams⟩
```

```
$ cat 2avg.exp
print((read + read) / 2)

$ krun 2avg.exp

5
7
6
⟨k⟩
  6
⟨/k⟩
⟨streams⟩
  ⟨in⟩
    .
  ⟨/in⟩
  ⟨out⟩
    .
  ⟨/out⟩
⟨/streams⟩
```

Figure 3.2 : l'exécution interactive avec **krun** des deux programmes EXP.

La configuration finale obtenue lors de l'exécution de p0.exp contient 15 dans la cellule k, dans le 2ème programme **krun** attend que l'utilisateur saisie les deux valeurs 5 et 7, afin d'afficher leurs moyenne.

II. Travaux de Vérification du BPEL

La vérification de processus BPEL a été le sujet de plusieurs travaux de recherche. La modélisation des processus BPEL, les outils utilisés pour la vérification et les propriétés considérées diffèrent d'un groupe à l'autre. Cette partie présente quelques travaux dans ce contexte.

1. Vérification du BPEL avec Spin/Promela

Dans un premier temps, nous présentons quelques travaux faits sur BPEL avec l'outil Spin/Promela.

- **Travaux de Fu, Bultan et Su**

Ce groupe de chercheurs s'est intéressé à la vérification de la composition des services Web (chorégraphie). Pour ce faire, il a proposé un modèle de composition qui consiste en la communication entre plusieurs pairs (un pair est un service Web) par l'intermédiaire de messages asynchrones. Chaque pair dispose d'une file d'attente FlFO (First ln First Out) pour le stockage des messages. L'ordre dans lequel les messages sont envoyés est enregistré par un observateur global virtuel. Ces séquences de messages observées représentent l'ensemble de conversations possibles pour cette composition. Donc, une composition de services Web est vue comme étant un ensemble de pairs disposant d'un « schéma de conversations» ayant un ensemble fini de types de messages [32].

Le passage du BPEL en spécification Promela demande de fournir l'ensemble des spécifications BPEL des processus et les WSDL contenant les déclarations des messages et des variables. Les types de messages sont extraits des fichiers WSDL. Chaque type de message (converti en MSL (Model Schema Language) qui est un modèle formel compact qui capture la plupart des structures de schéma XML dans le schéma de conversations) est traduit en Promela en un typedef définissant un enregistrement. Les chaînes de caractères qui regroupent les noms des messages et les symboles représentant l'état d'un automate sont présentés sous forme d'une énumération. Chaque activité est traduite en un automate gardé (guarded automata). Pour les activités sequence, switch et while, ils connectent l'état final à l'état initial des activités atomiques qu'elles contiennent. L'activité flow est construite en faisant le produit cartésien de toutes ces branches. Par la suite, chaque automate est traduit en processus Promela auquel ils assignent un canal de communication asynchrone.

Ainsi, pour chaque pair (service Web), ils déclarent un canal. La communication se fait par échange de messages dont le premier champ est le type de message envoyé. Dans la partie init du Promela, ils initialisent toutes les variables globales (peut être non-déterministe).

On trouve également dans init la création d'une instance de chaque processus. Pour exprimer les propriétés sur le schéma de conversations, ils utilisent la logique LTL. Mais puisque les files utilisées sont non bornées, ceci a rendu la vérification indécidable. Toutefois, en bornant ces files, ils ont pu utiliser les techniques de la vérification de modèles.

- Travaux de Nakajima

Dans un premier temps, ce chercheur a vérifié avec Spin la composition de services Web exprimée avec WSFL [33]. Par la suite, il a considéré BPEL. La traduction de BPEL s'effectue en deux parties; les activités sont représentées en automate fini étendu, puis l'automate est représenté en Promela. L'environnement avec lequel le processus interagit est représenté en Promela manuellement. En d'autres mots, le modèle Promela obtenu est fermé à la main [34].

Pour le traitement des flows, Nakajima a considéré les liens de connexions entre les activités. Lorsque l'activité source termine son exécution, elle assigne à chaque lien une valeur booléenne. Une activité est exécutée quand tous les liens entrants sont définis et sa condition de jointure (expression booléenne attribuée à l'activité) est vérifiée. Or, le vérificateur ne dispose d'aucune information sur l'environnement (aucun fournisseur de service, ni valeur concrète, ni message, n'est disponible). Par contre la vérification doit explorer toutes les possibilités; par exemple pour traiter une condition au niveau d'un lien source, il faut que les variables soient initialisées mais puisque ce n'est pas le cas, Nakajima a considéré que la condition peut être vraie comme elle peut être fausse. Dans certains cas, deux conditions complémentaires (une est complément de l'autre et vice versa) ne peuvent pas être vraies en même temps ce qui constitue un problème. Pour le surmonter, il a ajouté des variables de prédicats qui imposent une contrainte logique (par exemple, si une est vraie J'autre doit être fausse) [35].

Pour chaque type d'activité BPEL, une traduction en EFA (Extended Finite Automata) a été proposée. L'EFA résultant va ensuite être traduit en modèle Promela. Ce modèle est ensuite vérifié en utilisant la notion d'états qui ne peuvent pas être atteints (unreachable state pour l' interblocage) et en utilisant la logique LTL qui exprime des propriétés sur certains exemples.

- Travaux de Fernandez, Arias-Fisteus et Kloos

Cette équipe a conçu VERBUS (VERification for BUSiness processes) comme une application modulaire qui peut vérifier des processus d'affaire spécifiés avec différents langages et ce en utilisant différents outils de vérification. Le prototype qu'ils ont développé jusqu'à présent supporte Je BPEL comme langage de définition de processus et les outils Spin et SMV (Symbolic Model Verifier) comme outils de vérification [36].

VERBUS dispose d'une couche qui permet de produire un modèle du processus BPEL en traduisant ses activités en machines à états finis. Par la suite, elle compile ce modèle en un

langage connu par l'outil de vérification - Promela dans le cas de Spin et SMV dans le cas de SMV - pour qu'il soit traité par l'outil. Les propriétés à vérifier peuvent être exprimées en une notation d'expressions d'arbre et insérées dans la définition du processus BPEL. Elles peuvent également être définies en logique temporelle - LTL dans le cas de Spin et CTL dans le cas de SMV - et présentées à l'outil de vérification.

2. Vérification du BPEL avec divers autres outils

Dans ce qui suit, nous présentons quelques travaux faits sur BPEL en utilisant certains autres outils de vérification que Spin.

- **Travaux de Martens (réseaux de Petri)**

Ce chercheur s'est focalisé sur la notion d'utilisabilité des services Web: il vérifie si le service Web est utilisable avant de le mettre à la disposition des clients [37]. Pour cela, il a défini la notion de u-graph qui permet de voir si pour un module donné (service Web) il existe un environnement qui peut l'utiliser sans interblocage. Ayant un module et un environnement proposé compatibles syntaxiquement - leurs processus internes sont disjoints et chaque entrée pour l'un doit être une sortie pour l'autre - cet environnement peut utiliser correctement ce module s'ils sont cohérents, et ils le sont si à partir de chaque état de l'environnement on peut atteindre un état qui s'assortit avec un état final du module. Un module peut avoir plusieurs environnements possibles. L'environnement doit toujours envoyer les messages que le module est capable de consommer, et il doit considérer tous les choix possibles. Pour qu'il n'y ait pas d'interblocage, il faut généralement que l'environnement ait des informations sur l'état interne du module. À partir d'un algorithme proposé par ce chercheur, on peut construire le c-graph qui contient l'information maximale dont doit disposer un environnement par un module [37]. Ce chercheur veut générer l'environnement parfait qui représente tous les cas possibles pour un module donné. La composition de cet environnement et du module ne doit pas contenir d'interblocage ; d'une autre manière, le module (service Web) doit se terminer sans avoir besoin d'une interaction avec l'environnement.

- **Travaux de Foster (algèbres de processus)**

Dans sa thèse, Foster [23] a proposé l'outil LTSA-WS (Labelled Transition System Analyzer for Web Services), qui représente une extension de LTSA (qui est un outil de vérification de systèmes concurrents) pour vérifier des processus BPEL [38]. Son outil permet de vérifier si une composition de services Web implémentée en BPEL satisfait une spécification de cette composition présentée sous forme de MSC (Message Sequence Charts). Pour ce faire, il a traduit BPEL et MSC en notation d'algèbre de processus FSP (Finite State Process). Et pour

établir la vérification, il procède à la comparaison des deux représentations FSPs en vérifiant l'équivalence des traces. Cette vérification s'assure que le FSP du BPEL ne fournit pas des séquences non incluses dans l'ensemble des séquences du FSP du MSC et vice-versa. Foster a catégorisé les éléments du BPEL en quatre groupes pour faciliter le passage du BPEL en FSP. Les *eventHandlers* et les *correlations* n'ont pas été traités. De plus, un processus qui peut avoir plusieurs points de départ n'est pas supporté par l'outil [39].

3. Synthèse

La plupart de ces travaux de vérifications se basent sur la transformation des modèles, en transformons le code BPEL vers un autre formalise pour le vérifier. Ça peut impliquer sur la qualité de la sémantique transformée , ainsi sur la vérification. Dans ce travail, nous voulons développer une sémantique formelle basée sur la syntaxe du langage BPEL avec le Framework K. Ça sert à la fois un outil d'édition et de compilation du code BPEL ainsi un support puissant de sa vérification formelle.

III. Conclusion

Le Framework K démontre que la spécification formelle d'un langage de programmation peut être simultanément simple, expressif, analysable, évolutive et modulaire [30], permettant d'expérimenter facilement la conception des langages. Dans ce chapitre, nous avons montré quelque fonctionnalités offertes par l'outil K ainsi les différents travaux de recherches sur la vérification de processus BPEL. la meilleure façon de découvrir K Framework c'est de formaliser le maximum des langages de programmation ainsi de développer des nouveaux.

Chapitre 4 : Implémentation

Introduction :

Le présent chapitre présente la structure et les composants majeurs de la sémantique du langage BPEL dans K Framework , nous décrivons chacune des principales caractéristiques de ce langage ainsi que de fournir des extraits de leur mise en œuvre, de discuter de plusieurs fonctionnalités avancées du K Framework utilisées pour mettre en œuvre ces composants.

De point de vue formelle notre définition est en version initiale vu que c'est le premier travail basé sur K Framework pour spécifier la sémantique d'un langage de balisage comme BPEL, Ce qui est différent des langages impératifs comme C,C++, java et autres. Toutefois, notre grand objectif en fournissant cette sémantique c'est de fournir un outil pratique pour raisonner , manipuler et vérifier les programmes BPEL.

Les Modules de notre sémantique :

Notre définition du langage BPEL contient deux modules BPEL-SYNTAX et BPEL ,cette séparation de la syntaxe du reste de la définition est généralement considérée comme une bonne pratique et minimise l'ambiguïtés d'analyse.

I. Définition de la Syntaxe du BPEL

la syntaxe est définit en utilisant la notation BNF (Bakus-Naur Form), les terminaux sont bornés entre guillemets et les non-terminaux commençant par une lettre majuscule.

```
syntax Process     ::=
"<process name=" DescProcess ">" Extensions Imports PartnerLinks MessageExchanges Variables Correl
syntax DescProcess ::= PName TargetNamespace QueryLanguage ExpressionLanguage SuppressJoinFailure
                     | PName TargetNamespace Xmlns
                     | PName TargetNamespace QueryLanguage Xmlns
                     | PName TargetNamespace ExpressionLanguage Xmlns
                     | PName TargetNamespace SuppressJoinFailure Xmlns
                     | PName TargetNamespace ExitOnStandardFault Xmlns
                     | PName TargetNamespace QueryLanguage ExpressionLanguage Xmlns
                     | PName TargetNamespace QueryLanguage SuppressJoinFailure Xmlns
                     | PName TargetNamespace QueryLanguage ExitOnStandardFault Xmlns
```

Figure 4.1 Syntaxe d'un service BPEL

Un service BPEL peut faire usage d'extensions du langage : connexion à un système de gestion de base de données, accès au système de fichiers, appels de fonctions externes définies en Java, envoi d'e-mails... La syntaxe de déclaration d'une extension (Extension) est la suivante :

```
syntax Extensions ::= List{Extension,""}
syntax Extension  ::= "<extension name=" URI "mustUnderstand=" "yes" "/>"
                    | "<extension name=" URI "mustUnderstand=" "no" "/>"
                    |" "
```

Figure 4.2 Syntaxe de déclaration d'une extension

L'inclusion d'un fichier Xml Schema ou Wsdl se fait par l'intermédiaire d'une construction appelée Import qui a la syntaxe suivante :

```
syntax Imports ::= List{Import,""}
                 | " "
syntax Import ::= "<import name=" URI "location="URI "importType=" URI "/>"
                | "<import name=" URI "importType=" URI"/>"
                | "<import location=" URI "importType=" URI"/>"
                |" "
```

Figure 4.3 Syntaxe d'import

```
syntax PartnerLinks ::= "<partnerLinks>" LpartnerLinks "</partnerLinks>"
                      | " "
syntax LpartnerLinks ::= List{PartnerLink,""}
syntax PartnerLink   ::=
"<partnerLink name=" PName "partnerLinkType=" QName "myRole=" PName "partnerRole=" PName InitializePartnerRole "/>"
  | "<partnerLink name=" PName "partnerLinkType=" QName "/>"
  | "<partnerLink name=" PName "partnerLinkType=" QName "myRole=" PName "/>"
  | "<partnerLink name=" PName "partnerLinkType=" QName "partnerRole=" PName "/>"
  | "<partnerLink name=" PName "partnerLinkType=" QName "partnerRole=" PName InitializePartnerRole PName "/>"
syntax InitializePartnerRole::= "initializePartnerRole=""yes"
                              |"initializePartnerRole=""no"
                              |" "
```

Figure 4.4 Syntaxe de déclaration d'un lien de communication ' partnerLink'

```
syntax Variables  ::= "<variables>" Lvariables "</variables>"
                    |" "
syntax Lvariables ::= List{Variable,""}
syntax  Variable  ::=
"<variable name=" VarName "messageType=" QName "type=" QName "element=" QName "/>"
    | "<variable name=" VarName "/>"
    | "<variable name=" VarName "messageType=" QName "/>"
    | "<variable name=" VarName "type=" QName "/>"
    | "<variable name=" VarName "element=" QName "/>"
    | "<variable name=" VarName "messageType=" QName "type=" QName "/>"
    | "<variable name=" VarName "messageType=" QName "element=" QName "/>"
```

Figure 4.5 Syntaxe de déclaration d'une variable

Un service Web BPEL peut se dupliquer en plusieurs exécutions parallèles pour traiter plusieurs requêtes simultanément. Lorsque plusieurs réplications du même service sont exécutées, le problème se pose d'identifier la réplication vers laquelle un message reçu doit être redirigé.

Le mécanisme de corrélation proposé dans BPEL consiste à identifier la bonne réplication à partir des données contenues dans le message. L'idée repose sur le principe que parmi ces données, certaines (par exemple, un numéro de client) identifient de manière unique le

dialogue entre la réplication et son partenaire. Une telle donnée est appelée corrélation. Parfois, une donnée unique n'est pas suffisante et plusieurs valeurs doivent être extraites du message reçu afin d'établir avec certitude la réplication à laquelle le message est destiné. On parle alors d'ensemble de corrélations.

```
syntax CorrelationSets  ::= "<correlationSets>" LcorrelationSets "</correlationSets>"
                          |" "
syntax LcorrelationSets ::= List{CorrelationSet,""}
syntax CorrelationSet   ::= "<correlationSet name=" PName "properties=" QNamelist "/>"
```

Figure 4.6 Syntaxe de correlationSet

```
syntax FaultHandlers ::= "<faultHandlers>" Lcatch "</faultHandlers>"
                       | "<faultHandlers>" CatchAll "</faultHandlers>"
                       |" "
syntax Lcatch ::= List{Catch,""}
syntax Catch ::=
"<catch faultName=" QName "faultVariable=" VarName "faultMessageType=" QName ">" Activity "</catch>"
    | "<catch faultName=" QName "faultVariable=" VarName "faultElement=" QName ">"  Activity "</catch>"
    | "<catch faultName=" QName "faultVariable=" VarName ">" Activity "</catch>"
    | "<catch faultName=" QName ">" Activity "</catch>"
    | "<catch faultVariable=" VarName ">" Activity "</catch>"
    | "<catch faultMessageType=" QName ">" Activity "</catch>"
    | "<catch faultElement=" QName ">"  Activity "</catch>"
    | " "
syntax CatchAll ::= "<catchAll>" Activity "</catchAll>"
                  |" "
```

Figure 4.7 Syntaxe de faultHandlers

```
syntax EventHandlers ::= "<eventHandlers>" LonEvent "</eventHandlers>"
                       | "<eventHandlers>" LonAlarm "</eventHandlers>"
                       |" "
syntax LonEvent  ::= List{OnEvent,""}
syntax OnEvent   ::=  "<onEvent" DesconEvent Correlations FromParts Scope "</onEvent>"
syntax DesconEvent ::=
"partnerLink=" PName "portType=" QName "operation=" PName "messageType=" QName "variable=" VarName "messageExchange=" PName ">"
    |"partnerLink=" PName "portType=" QName "operation=" PName "element=" QName "variable=" VarName "messageExchange=" PName ">"
    |"partnerLink=" PName  "operation=" PName ">"
    |"partnerLink=" PName  "portType=" QName "operation=" PName ">"
    |"partnerLink=" PName  "operation=" PName  "messageType=" QName ">"
    |"partnerLink=" PName  "operation=" PName  "element=" QName ">"
    |"partnerLink=" PName  "operation=" PName  "variable=" VarName ">"
    |"partnerLink=" PName  "operation=" PName  "messageExchange=" PName ">"
    |"partnerLink=" PName  "portType=" QName "operation=" PName  "messageExchange=" PName ">"
    |"partnerLink=" PName  "portType=" QName "operation=" PName  "variable=" VarName ">"
    |"partnerLink=" PName  "portType=" QName "operation=" PName  "messageType=" QName ">"
    |"partnerLink=" PName  "portType=" QName "operation=" PName  "element=" QName ">"
    |"partnerLink=" PName  "portType=" QName "operation=" PName  "element=" QName ">"
    |"partnerLink=" PName  "operation=" PName  "messageType=" QName "variable=" VarName ">"
    |"partnerLink=" PName  "operation=" PName  "messageType=" QName "messageExchange=" PName ">"
    |"partnerLink=" PName  "operation=" PName  "element=" QName "messageExchange=" PName ">"
    |"partnerLink=" PName  "operation=" PName  "element=" QName "variable=" VarName ">"
    |"partnerLink=" PName  "operation=" PName  "variable=" VarName "messageExchange=" PName ">"
```

Figure 4.8 Syntaxe de définition d'un gestionnaire d'événement 'onEvent'

```
syntax Correlations ::= "<correlations>"  Linitiate "</correlations>"
                      |" "
syntax Linitiate ::= List{Initiate,""}
syntax Initiate ::= "<correlation set="PName "initiate=" "Yes" ">"
                   |"<correlation set="PName "initiate=" "join" ">"
                   |"<correlation set="PName "initiate=" "no" ">"
syntax FromParts ::= "<fromParts>" LFromParts "</fromParts>"
                    |" "
syntax LFromParts   ::= List{FromPart,""}
syntax FromPart ::="<fromPart part=" PName "toVariable=" VarName ">"
syntax LonAlarm ::= List{OnAlarm,""}
                   |" "
syntax OnAlarm ::=
    "<onAlarm" "<for expressionLanguage=" URI ">" Exp "</for>" Scope "</onAlarm>"
    | "<onAlarm" "<until expressionLanguage=" URI ">" Exp "</until>" Scope "</onAlarm>"
    | "<repeatEvery" "<until expressionLanguage=" URI ">" Exp "</repeatEvery>" Scope "</onAlarm>"
```

Figure 4.9 Syntaxe de définition d'un gestionnaire d'événement 'onAlarm'

```
syntax Activity ::= Receive
                  | Reply
                  | Invoke
                  | Assign
                  | Throw
                  | Exit
                  | Wait
                  | Empty
                  | Sequence
                  | If
                  | While
                  | RepeatUntil
                  | ForEach
                  | Pick
                  | Flow
                  | Scope
                  | Compensate
                  | CompensateScope
                  | Rethrow
                  | Validate
                  | ExtensionActivity
```

Figure 4.10 Syntaxe des activités

```
syntax Wait ::=
 "<wait" StandardAttributes StandardElements "<for expressionLanguage=" URI ">" Exp "</for>" "</wait>"
           |"<wait" StandardAttributes  "<for expressionLanguage=" URI ">" Exp "</for>" "</wait>"
           |"<until expressionLanguage="URI ">" Exp "</until>" "</wait>"
```

Figure 4.11 Syntaxe de l' activité wait

```
syntax Empty ::= "<empty" StandardAttributes StandardElements "</empty>"
               |"<empty" StandardAttributes "</empty>"
```

Figure 4.12 Syntaxe de l' activité empty

```
syntax Assign   ::="<assign validate=" Validate StandardAttributes StandardElements LCopy "</assign>"
                  |"<assign" StandardAttributes StandardElements LCopy "</assign>"
                  |"<assign" StandardAttributes LCopy "</assign>"
syntax Validate ::="validate=" "yes"
                  |"validate=" "no"
syntax StandardAttributes ::="name=" PName "suppressJoinFailure=" "yes" ">"
                            |"name=" PName "suppressJoinFailure=" "no" ">"
                            |"name=" PName ">"
                            |">"
syntax LCopy ::= List{Copy,""}
syntax Copy  ::= "<copy>" From To "</copy>"
                |"<copy keepSrcElementName=" KSEName ">" From To "</copy>"
                |"<copy ignoreMissingFromData=" IMFromData ">" From To "</copy>"
                |"<copy keepSrcElementName=" KSEName "ignoreMissingFromData=" IMFromData ">" From To "</copy>"
syntax From  ::=
"<from partnerLink="PName "endpointReference=" PName "/>"
 | "<from variable=" VarName "part=" PName ">" "<query queryLanguage=" URI ">" QueryContent "</query>" "</from>"
 | "<from variable=" VarName ">" "<query queryLanguage=" URI ">" QueryContent "</query>" "</from>"
 | "<from variable="VarName "part="  PName "/>"
 | "<from variable=" VarName  "property=" QName "/>"
 | "<from expressionLanguage=" URI ">" Exp"</from>"
 | "<from>" "<literal>" Literal "</literal>" "</from>"
syntax To  ::=
 "<to variable=" VarName "part=" PName ">" "<query queryLanguage=" URI ">" QueryContent "</query>" "</to>"
            |"<to variable="VarName "part=" PName "/>"
            |"<to variable="VarName "/>"
            | "<to variable=" VarName  "property=" QName "/>"
            | "<to expressionLanguage=" URI ">" Exp "</to>"
            | "<to partnerLink=" PName "/>"
```

Figure 4.13 Syntaxe de l' activité assign

```
syntax Receive  ::= "<receive" Descreceive StandardElements Correlations FromParts "</receive>"
                  | "<receive" Descreceive Correlations FromParts "</receive>"
                  | "<receive" Descreceive "</receive>"
syntax Descreceive ::=
"partnerLink=" PName "portType=" PName "operation=" PName "variable=" VarName CreateInstance "messageExchange=" PName StandardAttributes
    |"partnerLink=" PName "operation=" PName StandardAttributes
    | StandardAttributes CreateInstance "partnerLink=" PName "operation=" PName
    |"partnerLink=" PName "portType=" PName "operation=" PName StandardAttributes
    |"partnerLink=" PName "operation=" PName "variable=" VarName StandardAttributes
    |"partnerLink=" PName  "operation=" PName  CreateInstance StandardAttributes
    |"partnerLink=" PName   "operation=" PName "messageExchange=" PName StandardAttributes
    |"partnerLink=" PName "portType=" PName "operation=" PName "variable=" VarName StandardAttributes
    |"partnerLink=" PName "portType=" PName "operation=" PName  CreateInstance  StandardAttributes
    |"partnerLink=" PName "portType=" PName "operation=" PName  "messageExchange=" PName StandardAttributes
    |"partnerLink=" PName "operation=" PName "variable=" VarName CreateInstance  StandardAttributes
    |"partnerLink=" PName "operation=" PName "variable=" VarName  "messageExchange=" PName StandardAttributes
    |"partnerLink=" PName  "operation=" PName CreateInstance "messageExchange=" PName StandardAttributes
syntax CreateInstance ::= "createInstance=" "yes"
                        | "createInstance=" "no"
syntax StandardElements ::= Targets Sources
                          |" "
syntax Targets ::= "<targets>" "<joinCondition" "expressionLanguage=" URI ">" Exp "</joinCondition>"  LTarget "</targets>"
                  |"<targets>" LTarget "</targets>"
                  |" "
syntax LTarget ::= List{Target,""}
syntax Target ::="<target linkName=" TName "/>"
syntax Sources ::="<sources>" LSources "</sources>"
                  |" "
syntax LSources ::= List{Source,""}
syntax Source ::= "<source linkName=" TName "<transitionCondition" "expressionLanguage=" URI ">" Exp "</transitionCondition>" "</source>
                 |"<source linkName=" TName "</source>"
```

Figure 4.14 Syntaxe de l' activité receive

```
syntax Sequence ::=
  "<sequence" StandardAttributes StandardElements Activities "</sequence>"
            |"<sequence" StandardAttributes  Activities "</sequence>"
syntax Activities ::= Activity
              | Activities Activities    [left, strict]
```

Figure 4.15 Syntaxe de l' activité sequence

```
syntax If ::=
 "<if" StandardAttributes StandardElements Condition Activity Elseif Else "</if>"
    | "<if" StandardAttributes StandardElements Condition Activity  Else "</if>"
    | "<if" StandardAttributes StandardElements Condition Activity Elseif "</if>"
    | "<if" StandardAttributes StandardElements Condition Activity  "</if>"
    | "<if" StandardAttributes Condition Activity Elseif Else "</if>"
    | "<if" StandardAttributes Condition Activity  Else "</if>"
    | "<if" StandardAttributes Condition Activity Elseif "</if>"
    | "<if" StandardAttributes Condition Activity  "</if>"
syntax Condition ::= "<condition" "expressionLanguage=" URI ">" Exp "</condition>"
syntax Else ::= "<else>" Activity "</else>"
syntax LElseif ::= List{Elseif,""}
        |" "
syntax Elseif ::= "<elseif>" Condition Activity "</elseif>"
```

Figure 4.16 Syntaxe de l' activité if

```
syntax While ::=
 "<While" StandardAttributes StandardElements Condition Activity "</while>"
            |"<While" StandardAttributes Condition Activity "</while>"
```

Figure 4.17 Syntaxe de l' activité while

```
syntax RepeatUntil ::=
 "<repeatUntil" StandardAttributes StandardElements Activity Condition "</repeatUntil>"
              |"<repeatUntil" StandardAttributes Activity Condition "</repeatUntil>"
```

Figure 4.18 Syntaxe de l' activité repeatUntil

```
syntax Pick ::=
 "<pick createInstance=" "yes" StandardAttributes StandardElements OnMessage OnAlarm "</pick>"
 |"<pick createInstance=" "no" StandardAttributes StandardElements OnMessage OnAlarm "</pick>"
 |"<pick createInstance=" "yes" StandardAttributes OnMessage OnAlarm "</pick>"
 |"<pick createInstance=" "no" StandardAttributes OnMessage OnAlarm "</pick>"
syntax OnMessage ::=LDescOnMessage Correlations FromParts Activity "</onMessage>"
syntax LDescOnMessage ::=  List{DescOnMessage,""}
syntax DescOnMessage ::=
 "<onMessage partnerLink=" PName "portType=" PName "operation=" PName "variable=" VarName "messageExchange=" PName ">"
 |"<onMessage partnerLink=" PName "operation=" PName ">"
 |"<onMessage partnerLink=" PName "operation=" PName "variable=" VarName ">"
 |"<onMessage partnerLink=" PName "operation=" PName "portType=" PName ">"
 |"<onMessage partnerLink=" PName "portType=" PName "operation=" PName ">"
 |"<onMessage partnerLink=" PName "portType=" PName "operation=" PName "variable=" VarName ">"
 |"<onMessage partnerLink=" PName "portType=" PName "operation=" PName "messageExchange=" PName ">"
 |"<onMessage partnerLink=" PName "operation=" PName "variable=" VarName "messageExchange=" PName ">"
```

Figure 4.19 Syntaxe de l' activité pick

II. Sémantique du BPEL :

Un exemple d'une stratégie d'évaluation dans la déclaration de non-terminal *Activities* ,

```
syntax Activities ::= Activity
                    | Activities Activities    [left, strict]
```

Configuration

```
configuration <T color="red">
                <threads color="orange">
                  <thread multiplicity="*" color="yellow">
                    <k color="green"> ($PGM:Process) </k>
                     <state color="green">.Map </state>
                    <br/>
                    <control color="cyan">
                      <fstack color="blue"> .List </fstack>
                      <xstack color="purple"> .List </xstack>
                    </control>
                    <br/>
                    <env color="violet"> .Map </env>
                    <holds color="black"> .Map </holds>
                    <id color="pink"> 0 </id>
                  </thread>
                </threads>
                <br/>
                <genv color="pink"> .Map </genv>
                <store color="white"> .Map </store>
                <busy color="cyan">.Set</busy>
                <terminated color="red"> .Set </terminated>
                <br/>
                <in color="magenta" stream="stdin"> .List </in>
                <out color="brown" stream="stdout"> .List </out>
                <nextLoc color="gray"> 0 </nextLoc>
              </T>
```

Figure 4.20 Définition d'une configuration

Les règles (*Rules*) :

```
rule <k> X:Val=>I ...</k> <state>... X|-> I...</state>
rule Act1:Activity Act2:Activity => Act1 ~> Act2
rule  <process name=  DescProcess  >  Extensions Imports PartnerLinks MessageExchanges Variables
     => <process name=  DescProcess  >  Extensions Imports PartnerLinks MessageExchanges Variable
rule PName TargetNamespace QueryLanguage ExpressionLanguage SuppressJoinFailure ExitOnStandardFa
    => PName TargetNamespace "QueryLanguage" ExpressionLanguage SuppressJoinFailure ExitOnStanda
rule  <receive  Descreceive StandardElements Correlations FromParts  </receive>
    =>  <receive  Descreceive StandardElements Correlations FromParts  </receive>
rule  <assign validate=  Validate StandardAttributes  StandardElements LCopy  </assign>
  => <assign validate=  Validate StandardAttributes StandardElements LCopy  </assign>  [structural]
```

Figure 4.21 Exemple de règles

III. Compilation et exécution

L'outils K

Pour installer l'outil K suivez les étapes suivantes:

1. Télécharger la dernière version recommandé ou stable du http://www.kframework.org
2. Décompressez l'archive téléchargé. Cela va créer un dossier contenant k de l'ensemble de la distribution K.
3. Réglez votre variable d'environnement PATH pour le répertoire k / bin.

NB :il existe aussi une interface en ligne ainsi qu' une machine virtuelle KVM pour utiliser K Framework ;l'installation sur un système Unix (Linux) est aussi possible.

```
D:\TESTPFE\TEST\assign>more assign.bpel
<assign name= "PrepareInputForAAandDA">
<copy>
<from variable= "TravelRequest" part= "flightData"/>
<to variable= "FlightDetails" part= "flightData"/>
</copy>
<copy>
<from variable= "EmployeeTravelStatusResponse" part= "travelClass"/>
<to variable= "FlightDetails" part= "travelClass"/>
</copy>
</assign>
```

Figure 4.22: Exemple d'une activité assign.

```
D:\TESTPFE\TEST\assign>kompile bpel.k

D:\TESTPFE\TEST\assign>krun assign.bpel
<T>
    <k>
        <assign (name= "PrepareInputForAAandDA" >> (((copy> (
          <from variable= "TravelRequest" part= "flightData" />> (
          <to variable= "FlightDetails" part= "flightData" />> </copy>> (
          <copy> (<from variable= "EmployeeTravelStatusResponse" part=
          "travelClass" />> (<to variable= "FlightDetails" part= "travelClass"
           />> </copy>>> </assign>
    </k>
    <state>
        .Map
    </state>
</T>
```

Figure 4.23 : Exécution de l'activité assign avec la commande krun.

```
D:\TESTPFE\TEST\assign>krun receive.bpel
<T>
    <k>
        <receive (partnerLink= "PartnerLinkClient" portType=
          "tns:HelloWorldInvokeActivityBPELPortType" operation= "makeHello"
          variable= "input" (name= "Receive1" >>> </receive>
    </k>
    <state>
        .Map
    </state>
</T>
```

Figure 4.24 : Exemple d'une activité receive.

```
D:\TESTPFE\TEST\assign>kompile bpel.k -verbose
Init                                                    =   156
Including file: D:\TESTPFE\TEST\assign\bpel.k
Including file: E:\Kframework\k-nightly\k\include\modules\substitution.k
Including file: E:\Kframework\k-nightly\k\include\modules\pattern-matching.k
Including file: E:\Kframework\k-nightly\k\include\builtins\k-equal.k
Including file: E:\Kframework\k-nightly\k\include\builtins\bool.k
Including file: E:\Kframework\k-nightly\k\include\k-prelude.k
Including file: E:\Kframework\k-nightly\k\include\builtins\builtins.k
Including file: E:\Kframework\k-nightly\k\include\builtins\int.k
Including file: E:\Kframework\k-nightly\k\include\builtins\array.k
Including file: E:\Kframework\k-nightly\k\include\builtins\float.k
Including file: E:\Kframework\k-nightly\k\include\builtins\string.k
Including file: E:\Kframework\k-nightly\k\include\builtins\id.k
Including file: E:\Kframework\k-nightly\k\include\io\tcp.k
Including file: E:\Kframework\k-nightly\k\include\builtins\random.k
Including file: E:\Kframework\k-nightly\k\include\builtins\counter.k
Including file: E:\Kframework\k-nightly\k\include\builtins\symbolic-k.k
Including file: E:\Kframework\k-nightly\k\include\autoinclude.k
Including file: E:\Kframework\k-nightly\k\include\io\uris.k
Including file: E:\Kframework\k-nightly\k\include\builtins\map.k
Including file: E:\Kframework\k-nightly\k\include\builtins\bag.k
Including file: E:\Kframework\k-nightly\k\include\builtins\set.k
Including file: E:\Kframework\k-nightly\k\include\builtins\list.k
Including file: E:\Kframework\k-nightly\k\include\io\io.k
Basic Parsing                                           =  1357
Preprocess                                              =   328
Checks                                                  =    31
File Gen Pgm                                            =   171
File Gen Def                                            =   297
Importing Files                                         =  2371
Parsing Configs                                         =  2200
Parsing Rules                                           = 84411
Generating equations for hooks                          =   390
Generating Maude file                                   =  1217
Cleanup                                                 =   203
Total                                                   = 93132

Number of Modules                                       =     2
Number of Sentences                                     =    28
Number of Productions                                   =   375
Number of Cells                                         =     3
```

Figure 4. 25 L'exécution de la commande kompile avec l'option verbose .

IV. Conclusion

Dans ce chapitre nous avons proposé une définition du langage BPEL avec K Framework. L'idée de développement de cette plateforme a été largement motivée par l'observation que, après plus de 40 années de recherche systématique dans la sémantique des langages de programmation, ce domaine a resté ouvert seulement à la communauté des chercheurs et aux créateurs des langages de programmation [43].

Un environnement idéal pour la définition des langages de programmation doit être simple et facile pour comprendre et apprendre , il ne devrait pas demander aux utilisateurs d'avoir une compréhension approfondie de la théorie [43],ainsi cette plateforme sert comme un support à la vérification formelle des langages des programmation proposés ou existants.

Conclusion Générale

Dans ce mémoire nous nous somme intéressé à la sémantique formelle du BPEL avec K Framework, nous avons commencé avec une introduction au langage BPEL suivi par un aperçu sur la logique de réécriture qui décrit naturellement le comportement des systèmes concurrents et le système Maude qui supporte une large de gamme de technique de vérification formelle, en arrivons au K Framework, basé sur la réécriture et qui est utilisé pour définir des langages ainsi de spécifier et de pouvoir manipuler des sémantiques de façon formelle suivi par des travaux de recherches sur la vérification de processus BPEL en arrivons à cette l'implémentation

BPEL est un langage orienté à la modélisation des processus qui utilisent les services web pour leur implémentation. Néanmoins, c'est un langage de bas niveau, proche des langages de programmation ce qui le rend difficile à comprendre.

Le pouvoir expressif de la logique de réécriture pour décrire naturellement et d'une manière intuitive le comportement des systèmes concurrents, permet de révéler toutes les actions qui peuvent se produire en parallèle, décrivant ainsi correctement le comportement de ces systèmes selon une sémantique de vraie concurrence. Les déductions faites sur l'évolution des états d'un système représentent les calculs dans le système concurrent.

Maude est un système de haute performance c'est un langage de spécification formelle implémentant correctement tous les concepts théoriques de la logique de réécriture.

L'introduction et le développement du K Framework , un outil de définition et d'analyse des langages de programmation basé sur la logique de réécriture et le système Maude a été largement motivée par le fait qu'un environnement idéal pour la définition des langages de programmation doit être simple et facile pour comprendre et apprendre.

La définition de cette sémantique avec K Framework est une initiative qui permet de raisonner, manipuler et vérifier les programmes BPEL.

Bibliographie

[1] : Christensen E., Curbera F.,Meredith G,.Weerawarana S., 'Web Services Definition Language (WSDL) 1.1', W3C Note, March 15, 2001.

[2]: 'Business Process Execution Language for Web Services', BEA, IBM, Microsoft, SAP and Siebel, Version 1.1, May 2003.

[3]: 'Web service business Process Execution Language Version 2.0 Specification', OASIS Standard, 11 April 2007.

[4]: Clark J., 'XSL Transformations (XSLT) Version 1.0 ', W3C Recommendation, November 16, 1999.

[5]: Kholladi M.N, 'Une Approche de transformation de la notation BPMN vers BPEL basée sur la transformation de graphe', mémoire de magister en informatique, Option : Génie logiciel, Université Mentouri – Constantine ,2009.

[6]: Kirchner C., Kirchner H., Vittek M.,'Designing constraint logic programming languages using computational systems', In: Principles and Practice of Constraint Systems: The Newport Papers, The MIT Press, pp. 133 160, 1995.

[7]: Meseguer J.,' Rewriting Logic Revisited', Illinois University of Urbana Champaign, USA, 2002.

[8]: Meseguer J.' Rewriting logic as a unified model of concurrency'. In Lecture Notes in Computer Science, Volume 458, pages 384-400. August 1990.

[9]: Meseguer J.,' Multiparadigm Logic Programming'. In: H. Kirchner and G. Levi (eds.), Proc. Third Int. Conf. on Algebraic and Logic Programming, LNCS 632, Springer-Verlag, pages 158-200, 1992.

[10]:Marti-Oliet N., Meseguer J.,'Rewriting logic as logical and semantic Framework'.Technical Report SRI-CSL-93-05, SRI International, Computer Science Laboratory, August 1993.

[11]: Meseguer J.,'Rewriting logic as a semantic framework for concurrency'. In Lecture Notes in Computer Science, editor, 7th International Conference on Concurrency Theory, volume 1119. Springer Verlag August 1996.

[12]: Stehr M.O., Meseguer J., Ölveckzy P.C., 'Rewriting logic as a unifying framework for Petri Nets. In Unifying Petri Nets (Advances in Petri Nets)', Volume 2128 of Lecture Notes in Computer Science, pages 250-304. Springer Verlag. 2001.

[13]: Meseguer J., Talcott C.,' Formals Foundations for Compositional Software Architectures'. Position Paper, OMG-DARPA-MCC Workshop on Compositional Software Architectures, 1997.
[14]: Belala F., Latreche F., Benammar M..'Vers l'Intégration des Propriétés non Fonctionnelles dans le Langage SADL'. Revue de la Nouvelle Technologie de l'Information RNTI-L-2, Cépaduès-Editions, 2ièmeConférence Francophone Sur les Architectures Logicielles CFP- CAL2008, pp.91-105, ISSN : 1764-1667, ISBN : 978.2.85428.826.1, Montréal, Canada, Mars 2008
[15]: Bouanaka C., Belala F., Choutri A.,'On Generating Tile System for a Software Architecture: Case of a Collaborative Application Session', In Proceeding of ICSOFT'2007 (the Second Conference on Software and Data Technologies),pp. 123-128, 2007
[16]: Marti-Oliet N., Meseguer J.,'Rewriting Logic: Roadmap and Bibliography', In Theoretical Computer Science, June 2001.
[17]: Benammar M., 'Une Approche Basée Architecture pour la Spécification Formelle des Systèmes Embarqués',Thèse Doctorat en Sciences ,Spécialité : Informatique,Université Mentouri – Constantine ,2011
[18]: Bruni, R., Meseguer J., 'Semantic Foundations for Generalized Rewrite Theories', Theoretical Computer Science 360,pp. 386-414, 2006.
[19]: Clavel M., Duran F., Eker S., Marti-Oliet N., Lincoln P., Meseguer J., Talcott C.,'Maude Manual'. Version 2.6 January 2011.
[20]: Ölveczky P. C., Meseguer J.,'Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems', In 3rd International Workshop on Rewriting Logic and its Applications (WRLA'00), Vol. 36 of Electronic Notes in Theoretical Computer Science. Elsevier 2000.
[21]: Ölveczky P. C.,'Real-Time Maude 2.3 Manual',Department of Informatics, University of Oslo, 2007.
[22]: Rosu, G.,'Programming language design: Lecture notes', CS322, Fall 2003, Technical Report UIUCDCSR-2003-2897, University of Illinois at Urbana-Champaign, Department of Computer Science (2003),
[23]: Ellison C., Rosu G.,'An executable formal semantics of C with applications',in POPL. ACM, 2012, pp. 533–544.
[24]: Guth D.,'A formal semantics of Python 3.3', Master's thesis, University of Illinois at Urbana-Champaign, July 2013.
[25]: Bogdanas D., 'K definition of Java 1.4,' 2013
[26]: Meredith P., Hills M., Rosu G., 'An executable rewriting

logic semantics of K-Scheme',in SCHEME, D. Dubé, Ed.Laval University TR DIUL-RT-0701, 2007, pp. 91–103.
[27]: Lazar D., 'K definition of Haskell 98', 2012.
[28]: Park D., 'K definition of Javascript',2013.
[29]: Ellison C.,Lazar D.,'K definition of the LLVM assembly language,' 2012 .
[30]: Serbănută T.F,Arusoaie A.,Lazar D., Ellison C., Lucanu D., Rosu G.,'The K Primer (version 2.5)', Technical Report, January 2012.
[31]: Bolzmann G.J . , ' The Spin Model Checker: Primer and Reference Manual ', Addison-Wesley Professional,2003.
[32] : Fu X., Bultan T., Su. J., 'Analysis of Interacting BPEL Web Services', In Proceedings of the 13th International Conference (WWW), pages: 621-630, New York,2004.
[33] :Nakajima. S... 'Verification of Web Service FJows with Model-Checking Techniques ', Proceedings of the 1" International Symposium on Cyber Worlds, pages: 378-386, Tokyo, Japan. IEEE, 2002
[34] : Nakajima S.,'Model-checking Behavioral Specification of BPEL Applications', In Proceedings of the International Workshop on Web Languages and Forma] Methods, vol. 15 l, no. 2 (juillet) of Electronic Notes in Theoretical Computer Science, pages: 89-105, New castle, UK. Elsevier. 2005
[35] Nakajima S. ' Lightweight Formai Analysis of Web Service Flows ',Progress in Informatics, vol. 1, no. 2 (novembre), pages: 57-76, 2005
[36] : Fisteus J.A. , Fernandez L. S.,Kloos C.D.,'Applying Model Checking to BPEL4WS Business Collaborations '. Proceedings of the 2005 ,ACM symposium on Applied computing, pages: 826-830, Santa Fe, New Mexico,2005
[37] : Martens A ,'On Compatibility of Web Services ', Petri Net Newsletter, vol. 65 (octobre), pages: 12-20, 2003
[38] : Foster H.. 'A Rigorous Approach To Engineering Web Service Compositions ', PhD thesis, Imperial College London, 2006
[39]: Foster H. , Uchitel S. , Magee l., Kramer l,'Model-based l8 1h Verification of Web Service Compositions '. IEEE International Conference on Automated Software Engineering. Montreal, QC, Canada: IEEE Computer Society Press, pages: 152-161, 2003
[40]: Coq development Team, 'The Coq Proof Assistant, Reference Manual', December 16,2013

[41]: Borras P.,Clément D.,Despeyroux Th.,Incerpi J.,Kahn .,Lang.,Pascual,V.,'CENTAUR:The system,', In proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Development Environments (PSDE),Volume 24,pages 14-24,New York,1989
[42]: Rosu, G.,'K Overview Presentation', University of Illinois at Urbana-Champaign', Department of Computer Science,2012
[43]: Rosu, G., Serbănută T.F,' An Overview of the K Semantic Framework', J.LAP, Volume 79(6), pp 397-434. 2010

Webographie

http://www.kframework.org/, consulté le 02 Juin 2014.
http://maude.cs.uiuc.edu/ , consulté le 02 Juin 2014.