UNIVERSITÉ DU QUÉBEC À MONTRÈAL

VÉRIFICATION DE PROCESSUS BPEL À L'AIDE DE PROMELA-SPIN

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

AIDA CHAMI

MARS 2008

À celui qui m'a toujours poussée et motivée en m'indiquant la bonne voie...

À mon Père

À celle qui m'a enseigné la volonté et la patience...

À ma Mère

REMERCIEMENTS

Je souhaite tout d'abord adresser ma profonde gratitude à mon directeur de recherche, Monsieur Guy Tremblay, professeur à l'Université du Québec à Montréal, pour l'appui constant qu'il m'a apporté et l'apprentissage qu'il m'a permis. Je le remercie, plus que toute autre personne, pour la disponibilité, la patience, et l'ouverture dont il a fait preuve, ainsi que pour le support financier qu'il m'a accordé durant la réalisation de cette recherche. Ce support financier provient de subventions de recherche accordées par le CRSNG du Canada.

J'aimerais remercier chaleureusement mon co-directeur de mémoire, Monsieur Aziz Salah, professeur à l'Université du Québec à Montréal, pour sa rigueur intellectuelle, la qualité des informations qu'il m'a transmises, son encouragement continu, ainsi que pour la pertinence de ses commentaires et suggestions. Enfin, je lui suis très reconnaissante pour le support financier qu'il m'a accordé.

Je tiens à remercier sincèrement l'Université du Québec à Montréal, en particulier, tous les professeurs de son département informatique qui m'ont enseigné durant ma maîtrise. Mes remerciements s'adressent particulièrement à Monsieur Ivan Maffenzzini pour ses conseils et son écoute. Mes remerciements s'adressent aussi aux professeurs qui ont accepté de corriger mon mémoire.

Enfin, je remercie les membres de ma famille qui ont porté ce projet avec moi, en particulier mes sœurs Leïla et Widad pour leur appui perpétuel, et mes frères Ahmed et Amine pour leurs encouragements.

TABLE DES MATIÈRES

LISTE DES FIGURES

ACRONYMES

Acronyme	Signification
BPEL	*Business Process Executable Language*
WSDL	*Web Services Description Language*
Promela	*Process Meta LAnguage*
UDDI	*Universal Description, Discovery and Integration*
WSFL	*Web Services Flow Language*
BDD	*Binary Decision Diagrams*
LTL	*Linear Temporal Logic*
CTL	*Computation Tree Logic*
FIFO	*First In First Out*
DPE	*Death Path Elimination*
EFA	*Extended Finite Automata*
VERBUS	*VERification for BUSiness processes*
MSC	*Message Sequence Charts*
FSP	*Finite State Process*
SPIN	*Simple Promela INterpreter*
DOM	*Document Object Model*

BMC	*Bounded Model Checking*
W3C	*World Wide Web Consortium*
BPWS4J	*Business Process Execution Language for Web Services Java Runtime*

DÉFINITIONS

Terme	Définition
Système	Une collection de composants organisés pour accomplir une fonction ou un ensemble de fonctions spécifiques.
Environnement	Une partie du monde avec laquelle un système va interagir.
Système déterministe	Un système dont les conditions initiales identiques et avec les mêmes stimulations conduisent à des évolutions identiques (le comportement des sorties est maîtrisé).
Système non-déterministe	Un système tel que dans au moins un de ses états, il n'a pas une opération précise à réaliser, mais il doit faire un choix parmi plusieurs (sans que ce choix soit nécessairement aléatoire).
Modèle	Représentation simplifiée du système étudié.
Modèle fermé	Un modèle qui inclut toutes les parties avec lesquelles il peut interagir.
Contre-exemple	Un cas particulier qui prouve qu'une propriété est fausse.

RÉSUMÉ

L'objectif de notre travail de recherche est de vérifier si un processus *BPEL* satisfait sa spécification d'interface représentant son comportement externe en utilisant la vérification de modèles. Dans ce mémoire, nous présentons essentiellement l'approche de notre logiciel qui permet dans un premier temps de traduire un processus *BPEL* en modèle *Promela* et une expression d'interface en assertion de traces, et par la suite, il lance la vérification en utilisant l'outil *Spin*. Cette vérification du comportement du processus concret se fait par rapport à une spécification abstraite de son interface comportementale, c'est-à-dire, nous vérifions uniquement ce qui est visible à l'exterieur du processus. Nous expliquons les étapes franchies pour atteindre notre objectif et nous montrons à l'aide d'exemples que notre logiciel est fonctionnel.

INTRODUCTION

Un processus d'affaire 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.

Vu l'évolution qu'a connue le réseau Internet, les entreprises veulent que leurs partenaires puissent accéder directement à leurs fonctionnalités. 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 d'affaire 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'objectif est de ne rendre public que certains aspects du processus, à savoir les messages qu'il envoie et reçoit, et non pas les détails de la mise en œuvre.

Pour s'assurer que cette dissimulation de l'information soit bien faite, nous avons proposé au niveau de notre travail de maîtrise de vérifier si un processus respecte son interface comportementale abstraite, c'est-à-dire, si sa mise en œuvre satisfait son interface. Donc, notre but est qu'à partir d'une description détaillée et opérationnelle du comportement d'un processus, on puisse détecter une exécution (comportement) qui ne respecte pas son interface à l'aide de la vérification de modèles. La description du processus est spécifiée en langage *BPEL* (*Business Process Executable Language*) qui est un langage proposé pour présenter un type de spécification des services Web de plus haut niveau que celui exprimé en WSDL (*Web Services Description Language*) qui décrit la syntaxe des messages envoyés ou reçus ainsi que les opérations qu'un service peut effectuer. Cette dernière spécification est nécessaire mais insuffisante, puisqu'on a aussi besoin de connaître l'ordre dans lequel les services s'échangent des messages entre eux, ce qui peut être décrit par le langage *BPEL*. En

d'autres mots, *BPEL* décrit les interactions qu'un processus peut établir avec d'autres processus (ses partenaires) pour réaliser un service précis.

La vérification de processus *BPEL* avec des outils de vérification de modèles est le sujet d'un nombre notable de travaux qui ont utilisé différentes techniques d'abstraction et d'outils de vérification. Dans notre travail, nous proposons une abstraction particulière et nous représentons en assertion de traces des expressions d'interface, représentant les actions possibles vues de l'extérieur du processus.

Nous avons développé un logiciel qui se compose de trois modules comme le montre la figure 0.1, dont deux que nous avons nous-même développés. Le premier module est l'abstracteur/traducteur *BPEL_to_Promela* qui permet de produire, à partir d'une spécification d'un processus *BPEL*, un modèle *Promela* (*Process Meta Language*). Cette partie repose sur une technique d'abstraction qui prend en considération les types de propriétés que l'on veut vérifier, et une technique de traduction qui permet de fournir l'équivalent de chacun des composants *BPEL* retenus par l'abstracteur. Le deuxième module, *Generation_Traces*, permet de générer une spécification en assertions de traces *Spin* à partir d'une spécification d'interface du processus. Le troisième module effectue ensuite la vérification. L'étape de vérification est effectuée à l'aide de la vérification de modèles (*model checking*), plus précisément, l'outil *Spin*.

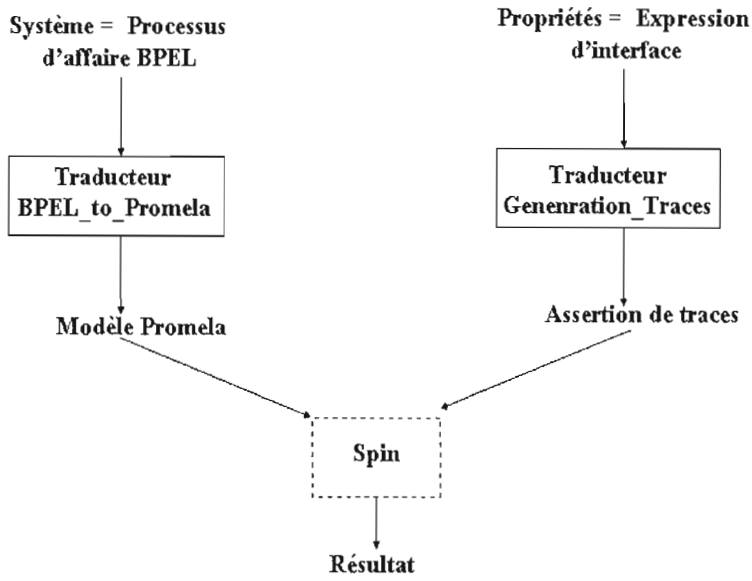


Figure 0.1 - Vue d'ensemble

En général, la vérification de modèles permet d'assurer le bon fonctionnement du système avant de se plonger dans sa mise en oeuvre. Elle joue un rôle important dans la détection de la source d'erreur puisque dans les cas où une propriété n'est pas satisfaite, elle renvoie une séquence d'exécution qui génère cette erreur. L'outil pour lequel nous avons opté pour effectuer cette vérification est *Spin* qui est utilisé pour analyser les systèmes distribués décrits en *Promela*, qui est un langage de construction de modèles disposant de primitives riches qui lui permettent de décrire le comportement de chacun des processus composant le système et leurs interactions.

Ce mémoire est organisé comme suit:

- Le premier chapitre présente les processus d'affaire, les services Web et le langage *BPEL.*

- Le deuxième chapitre explique ce qu'est la vérification de modèles et comment on peut effectuer la spécification de propriétés. Il présente également les propriétés que notre logiciel vérifie.

- Le troisième chapitre présente un certain nombre de travaux effectués dans le domaine de la vérification des processus d'affaires avec l'outil *Spin* ou avec d'autres outils de vérification.

- Le quatrième chapitre présente les outils et les technologies que nous avons utilisés pour mettre en œuvre notre logiciel.

- Le cinquième chapitre explique la mise en œuvre de notre outil de vérification.

- Le sixième chapitre présente un certain nombre de processus *BPEL* qui ont été testés par notre logiciel.

CHAPITRE I : SERVICES WEB ET PROCESSUS BPEL

1.1 Processus d'affaire et Services Web

Un processus d'affaire est un ensemble indépendant d'activités ordonnées où chaque activité effectue une fonctionnalité donnée. Certaines de ces activités peuvent être manuelles: c'est-à-dire exécutées par des employés jouant un rôle bien déterminé. Si la décision humaine est exigée dans un processus d'affaire, celui-ci ne peut pas être complètement automatisé, seulement certaines de ses portions.

Parmi Les mises en œuvre possibles d'un processus d'affaire, on trouve qu'il peut être vue comme étant un ensemble de services Web, puisqu'un service Web exécute une fonction qui commence par une simple demande/réponse et s'étend à un processus d'affaire complet [3].

Les services Web sont une technologie qui permet d'établir une communication à distance via le réseau Internet entre des applications qui reposent sur des plates-formes et des langages différents au moyen de protocoles universels standardisant les modes d'invocation.

La notion d'invoquer une application à distance n'est pas récente : l'innovation apportée par les services Web est la mise en place de standards (SOAP, WSDL, UDDI) [37] pour représenter tous les aspects du processus. Ceci a comme avantage la facilité de mise en œuvre ainsi que l'interopérabilité. De plus, les standards utilisés sont tous fondés sur XML, ce qui favorise leur cohésion.

Les services Web utilisent WSDL (*Web Service Description Language*) [6] qui permet de décrire les informations qui sont utiles pour invoquer des fonctions à distance par l'échange de messages (plus de détails dans la section suivante). Ils utilisent également UDDI (*Universal Description, Discovery and Integration*) [37] qui est un annuaire d'entreprises conçu comme un registre consultable classant les services Web disponibles avec leurs descriptions. Le rôle d'UDDI est de permettre aux clients de découvrir les services Web

qu'ils souhaitent exploiter en fournissant les informations permettant de les invoquer à distance et dynamiquement. Donc, UDDI favorise les communications entre clients et fournisseurs de services Web en facilitant la recherche et la découverte de ces services.

La mise en œuvre d'un service Web commence par la publication du service par son fournisseur auprès d'un distributeur. Lorsque le client a besoin d'un service, il effectue une recherche auprès du distributeur. Il reçoit alors une réponse qui est un document WSDL qui contient les informations concernant la localisation du service, la méthode d'invocation, les paramètres associés et le format de réponse. Par l'intermédiaire de ces informations, le client va pouvoir invoquer le service Web du fournisseur. La communication entre le client et le fournisseur se fera alors par échange de messages, décrits dans le document WSDL.

1.2 WSDL: Web Service Definition Language

Lorsqu'un client souhaite faire appel à un service Web pour répondre à certains besoins, il doit demander son fichier WSDL dans le but d'avoir des informations sur son emplacement, la méthode d'accès et les fonctions offertes. Le fichier WSDL définit les services Web comme une collection de points d'accès de service (`endpoints`) ou de ports susceptibles d'envoyer ou de recevoir des messages. WSDL repose sur une description abstraite des fonctionnalités du service en définissant l'ensemble des opérations et des messages. Il utilise une grammaire conforme à un schéma XML pour décrire de manière indépendante du langage et de la plate-forme la manière avec laquelle un service peut être accédé.

WSDL intègre les informations sur les fonctions disponibles publiquement, les types de données des messages, le protocole de transport à utiliser, l'adresse pour localiser le service, et les définitions abstraites des données. WSDL rend les services Web auto descriptifs puisqu'il permet de décrire comment les clients peuvent les invoquer.

Un document WSDL est composé d'un ensemble de définitions. Comme le montre la figure 1.1, un service est défini par un groupe de ports reliés entre eux. Un port définit les opérations qu'il peut réaliser et une opération est définie par les messages d'entrée et/ou les messages de sortie qu'elle reçoit ou transmet.

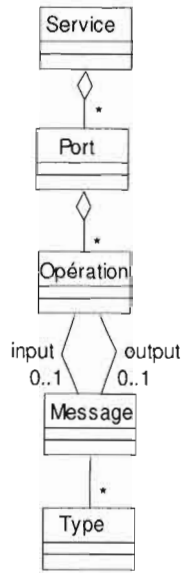


Figure 1.1 - Représentation de la partie abstraite d'un service WSDL

Un service Web est décrit en WSDL à l'aide de cinq éléments majeurs [6]. Dans ce qui suit, la majorité des extraits du code sont tirés de l'exemple de l'annexe A (page 98), mais aussi d'autres que nous avons créé nous-même:

- `Types` permet de définir des types en utilisant des schémas du langage XML. Ce sont des conteneurs de définitions de types de données.

```
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
             xmlns="http://www.w3.org/2000/10/XMLSchema">
         <element name="TradePriceRequest">
              <complexType>
                  <all>
                   <element name="tickerSymbol"
                            type="string"/>
                  </all>
              </complexType>
         </element>
           ...
  </schema>
</types>
```

- `Message` est utilisé pour fournir une définition typée abstraite des données communiquées. Un message consiste en une ou plusieurs parties (`part`), chacune étant associée à un type (simple ou complexe). Ces parties peuvent être vues comme des attributs du message utilisés pour décrire le contenu abstrait du message.

```
<message name="GetLastTradePriceInput">
      <part name="body" element="xsd:TradePriceRequest"/>
</message>
```

- `PortType` spécifie un ensemble d'opérations fournies par l'`endpoint`. À chaque opération peut être associé zéro ou un message d'entrée, et zéro ou un message de sortie.

```
<portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
         <input message="tns:GetLastTradePriceInput"/>
         <output message="tns:GetLastTradePriceOutput"/>
      </operation>
</portType>
```

  Il y a quatre sortes d'opérations, selon les messages associés :

  - *One-Way* : permet la réception d'un message par le service (`input`),
  - *Request-Response* : permet la réception d'un message et l'envoi d'un message comme réponse (`input`, `output`),
  - *Solicit-Response* : permet l'envoi d'un message qui est suivi par la réception d'un autre message comme réponse (`output`, `input`), et
  - *Notification* : permet l'envoi d'un message (`output`).

Ce qui a été présenté jusqu'à maintenant représente la description du niveau abstrait. Ce qui va suivre représente l'aspect concret de WSDL, qui n'est pas pris en compte dans notre travail mais que nous présentons pour que notre explication soit complète.

- `Binding` est une entité qui décrit la manière concrète par laquelle le service sera implémenté en oeuvre. Il inclut le protocole de communication utilisé, et le format

des données pour les opérations et les messages déterminés par un `portType` spécifique. Il ne donne aucune information sur l'adresse du service.

```
<binding name="StockQuoteSoapBinding"
         type="tns:StockQuotePortType">
      <soap:binding style="document"
           transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="GetLastTradePrice">
         <soap:operation
           soapAction="http://example.com/GetLastTradePrice"/>
         <input>
             <soap:body use="literal"/>
         </input>
         <output>
             <soap:body use="literal"/>
         </output>
      </operation>
</binding>
```

- `Service` combine un groupe de ports, lesquels sont définis comme des combinaisons d'une adresse réseau et d'un `binding`. Il définit donc les adresses qui peuvent être utilisées pour invoquer le service Web en question.

```
<service name="StockQuoteService">
     <documentation>My first service</documentation>
     <port name="StockQuotePort"
               binding="tns:StockQuoteBinding">
          <soap:address
               location="http://example.com/stockquote"/>
     </port>
</service>
```

1.3 BPEL4WS : Business Process Execution Language for Web Services

BPEL4WS (raccourci *BPEL*) est un langage défini par une grammaire XML qui spécifie l'aspect comportemental d'un processus d'affaire en décrivant la logique exigée pour coordonner les services Web qui participent dans le flux (`Flow`) de ce processus. *BPEL* a été conçu à partir du langage WSFL (*Web Services Flow Language*) d'IBM et du langage XLANG (*Web Services for Business Process Design*) de Microsoft. *BPEL* utilise des activités structurées – comme des activités séquentielles, parallèles, conditionnelles, et boucles – pour construire la logique d'affaire d'un processus. Un processus est spécifié par ses interactions avec ses partenaires. Un processus et ses partenaires sont décrits avec WSDL.

Un processus *BPEL* représente tous les partenaires et les interactions avec ces partenaires en terme d'interfaces WSDL abstraites (`portTypes` et `operations`).

BPEL permet de décrire tant des processus d'affaire abstraits que des processus exécutables. Un processus *BPEL* abstrait indique les échanges publics de messages entre les parties ; il n'est pas exécutable et ne donne pas de détails internes. Par contre, un processus *BPEL* exécutable modélise le déroulement des opérations et inclut les détails internes. *BPEL* ne permet pas de décrire la chorégraphie qui est une description globale, et non pas centralisée, de la collaboration d'un ensemble de services Web afin d'accomplir une tâche bien déterminée [2], puisqu'il donne toujours la description du point de vue d'un seul processus.

Comme le montre la figure 1.2, une définition d'un processus *BPEL* consiste dans les éléments suivants :

- Les déclarations des éléments qui vont être utilisés par ce processus – tels `partnerLinks`, `variables`, etc. – éléments qui seront manipulés par le processus au niveau de différentes interactions avec ses partenaires,

- La description du comportement du processus (`workflow`). Pour cela, *BPEL* utilise des activités primaires pour exécuter des opérations de base et des activités structurées – par exemple, activités séquentielles, concurrentes, etc. – pour définir l'ordre des activités de base.

BPEL fournit aussi un autre type d'activités pour gérer les erreurs d'une manière globale ou locale (dans des `scopes` qui seront présentés à la fin de ce chapitre).

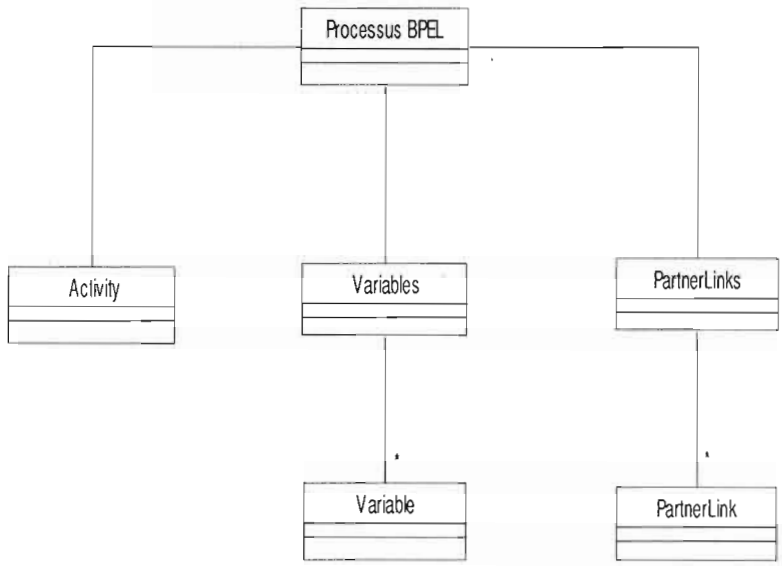


Figure 1.2 - Les Composants d'un processus BPEL

Un document *BPEL* contient les éléments suivants [1] – la majorité des extraits de code sont tirés des exemples des annexes B, C, D et E (pages 100 – 111):

- `PartnerLinks` : les services avec lesquels le processus interagit sont modélisés comme des `partnerLinks`. Leurs déclarations déterminent la forme statique des relations que le processus aura avec d'autres processus. Un `partnerLink` représente donc une relation de conversation entre deux processus collaborateurs.

```
<partnerLinks>
      <partnerLink name="client"
                   partnerLinkType="tns:LoanFlow"/>
        ...
</partnerLinks>
```

Chaque `partnerLink` est caractérisé par un `partnerLinkType` qui spécifie le rapport interactif entre deux services en déterminant les rôles qu'ils jouent, et le `portType` sur lequel chaque service va recevoir des messages au niveau d'une conversation donnée. Avec la capacité d'extension du WSDL 1.1, on place la

définition des `partnerLinkTypes` au niveau du fichier WSDL comme élément fils de l'élément `<WSDL:definition>`.

```
<plnk:partnerLinkType name="ncname">
      <plnk:role name="ncname1">
           <plnk:portType name="qname1"/>
      </plnk:role>
      <plnk:role name="ncname2">
           <plnk:portType name="qname2"/>
      </plnk:role>
</plnk:partnerLinkType>
```

- `Variables` : elles permettent à un processus *BPEL* d'avoir un état interne. Dans l'exemple ci-dessous, le type de la variable est spécifié avec l'attribut `messageType`.

```
<variables>
      <variable name="input"
                messageType="tns:LoanFlowRequestMessage"/>
      <variable name="sessionMsg"
                 messageType="tns:sessionMessage"/>
      ...
</variables>
```

Dans certains cas, le type de la variable peut être spécifié avec l'attribut `type` ou l'attribut `element` comme le montre l'exemple ci-dessous.

```
<variables>
      <variable name="var1"
                type="string"/>
      <variable name="var2"
                 element="e:StatusContainer"/>
      ...
</variables>
```

Les variables peuvent être manipulées par des activités d'affectation (`assign`) ou par des activités de réception ou d'envoi de messages.

```
<assign>
  <copy>
      <from variable="input" part="payload"/>
      <to variable="loanApplication" part="payload"/>
  </copy>
</assign>
```

BPEL permet aussi de déclarer des variables locales, déclarées au niveau des `scopes`.

- `CorrelationSets` : un processus *BPEL* supporte une ou plusieurs conversations (échanges de messages) avec ses partenaires durant son cycle de vie. Les messages envoyés doivent être livrés au port de destination correspondant ainsi qu'à la bonne instance du processus. Pour cette raison, *BPEL* offre un mécanisme déclaratif pour spécifier des groupes d'opérations corrélées dans une instance donnée. Un `correlationSet` est un groupe de propriétés qui permettent d'identifier une instance unique du processus. Un ensemble de corrélations est employé dans les activités `receive`, `reply`, `invoke` et `onMessage`. On peut avoir des définitions de ces ensembles dans le processus d'une manière globale ou dans les `scopes` du processus d'une manière locale. Lorsqu'un ensemble de corrélations est initialisé, les valeurs des propriétés doivent être semblables pour tous les messages dans toutes les opérations qui supportent cet ensemble. Un message peut supporter (être inclus dans) plusieurs ensembles de corrélations.

```
<correlationSets>
   <correlationSet name="auctionIdentification"
                   properties="as:auctionId"/>
</correlationSets>
```

- `FaultHandlers` : un `faultHandler` est composé d'activités `catch`, dont chacune permet de capturer une sorte spécifique d'erreurs. On peut également avoir une clause `catchAll` qui peut attraper toutes les erreurs non interceptées par les autres `catchs`. Une erreur est définie par un nom unique et une variable pour la donnée associée à cette erreur. La capture d'erreurs se fait par le nom de l'erreur s'il est présent, sinon par le type de la variable de l'erreur, mais cette variable reste optionnelle. Le `scope` dans lequel l'erreur se produit se termine d'une manière anormale que l'erreur soit capturée et traitée ou non. Si l'erreur se produit dans le `scope` global du processus et n'est associée à aucun `faulthandler` alors le processus se termine d'une manière anormale. *BPEL* permet également de signaler les erreurs à l'intérieur du processus d'une manière explicite et cela en utilisant

l'activité `throw` qui doit fournir le nom de l'erreur et peut optionnellement spécifier la valeur associée à cette erreur.

```
<faultHandlers>
      <catch faultName="lns:cannotCompleteOrder"
             faultVariable="POFault">
             ...
      </catch>
</faultHandlers>
```

- `EventHandlers` : on peut associer à un processus un ensemble d'`eventHandlers` qui sont invoqués concurremment à l'activité normale du processus si l'événement correspondant se produit. Ces `eventHandlers`, contrairement aux `fanltHandlers`, sont considérés comme une partie normale du processus. L'événement peut être soit un message d'entrée (opération requête/réponse ou opération d'une seule direction), soit une alarme. Le corps des `eventHandlers` peut contenir des activités de n'importe quel type. Les `eventHandlers` ne sont accessibles qu'après la création d'instances du processus. Ils peuvent se produire à des périodes irrégulières et un nombre arbitraire de fois tant que le `scope` correspondant est actif.

```
<eventHandlers>
       <onMessage partnerLink="canal1"
                  portType="port1"
                  operation="operation1"
                  variable="variable1"
                  name="name1">
          ...
       </onMessage>
       <onAlarm for="'PT2M'">
          ...
       </onAlarm>
</eventHandlers>
```

1.4 Les activités du processus BPEL

Le comportement du processus *BPEL* est décrit en utilisant des activités. Ces activités peuvent être catégorisées comme activités de base et activités structurées.

1.4.1 Les activités de base

Les opérations fournies par les partenaires ou les opérations fournies par le processus peuvent évidemment font partie des activités d'un processus. L'appel peut être synchrone (requête/réponse) ou asynchrone (une seule direction). Pour le cas synchrone, l'opération invoquée peut retourner un message d'erreur WSDL. Ces appels et exécutions d'opérations sont effectués via les activités `invoke`, `receive` et `reply`.

L'activité `receive` joue un rôle clé dans le cycle de vie d'un processus d'affaire, puisqu'elle permet d'initialiser le processus si son attribut `createInstance` le signale. On peut avoir plusieurs activités initiales concurrentes, dans ce cas l'activité qui arrive la première crée l'instance du processus.

```
<receive name="receiveInput"
         partnerLink="client"
         portType="tns:LoanFlow"
         operation="initiate" variable="input"
         createInstance="yes"/>
```

L'activité `reply` permet d'envoyer une réponse à une demande faite par `receive`, et correspond à un type de communication synchrone. Cette activité a deux formes possibles : soit la réponse est normale, soit une erreur est présente. Si une erreur est présente, `reply` envoie le nom de l'erreur avec la variable correspondant à l'erreur.

```
<reply partnerLink="customer"
       portType="lns:purchasePT"
       operation="sendPurchaseOrder"
       variable="Invoice"/>
```

L'activite `invoke` permet au processus d'exécuter (d'appeler) l'opération d'un partenaire d'une manière unidirectionnelle (`one-way`) ou demande-réponse (`request-response`).

```
<invoke name="invokeCR"
        partnerLink="creditRatingService"
        portType="services:CreditRatingService"
        operation="process"
        inputVariable="crInput"
        outputVariable="crOutput"/>
```

Dans le cas où le `invoke` retourne une réponse (présence d'`outputVariable`), cette réponse peut être aussi un message d'erreur.

```
<invoke partnerLink="creditRatingService"
        operation="process"
        inputVariable="crInput"
        outputVariable="crOutput">
    <catch faultName="nsxml0:NegativeCredit"
           faultVariable="crFault">
            <assign>
               <copy>
                   <from expression="20"/>
                   <to variable="crOutput"
                       part="payload"/>
               </copy>
            </assign>
    </catch>
</invoke>
```

1.4.2 Les activités structurées

Les activités structurées sont les suivantes :

- Contrôle séquentiel entre activités, fourni par `sequence`, `switch` et `while`. Une activité `sequence` peut contenir plusieurs activités qui sont exécutées séquentiellement.

```
<sequence>
   <invoke name="invokeUnitedLoan"
           partnerLink="UnitedLoanService"
           portType="services:LoanService"
           operation="initiate"
           inputVariable="loanApplication"/>
   <receive name="receive_invokeUnitedLoan"
            partnerLink="UnitedLoanService"
            portType="services:
                        LoanServiceCallback"
            operation="onResult"
            variable="loanOffer1"/>
</sequence>
```

Une activité `switch` est une activité conditionnelle. Cette activité consiste en une liste ordonnée d'une ou de plusieurs branches conditionnelles définies par des éléments `case`, suivies d'une branche optionnelle `otherwise`.

L'activité `while` permet l'exécution répétitive d'une activité spécifiée tant qu'une condition booléenne est vérifiée.

```
<while
      condition="bpws:getVariableData('var')">
                ...
</while>
```

- Concurrence et synchronisation entre activités avec `flow`. Les activités concurrentes sont représentées par `flow` et la synchronisation entre ces activités peut être assurée par le constructeur `link` qui permet d'exprimer une dépendance entre les activités. Un `flow` complète son exécution si toutes les activités qu'il contient complètent aussi leur exécution.

- Choix basé sur les évènements externes fournis par `pick`. L'activité `pick` attend l'occurrence d'un évènement parmi une liste d'évènements possibles. Une activité est associée à chaque évènement. L'activité exécutée est celle de l'évènement qui survient le premier. Si les évènements surviennent « simultanément » alors le choix de l'activité dépend de la mise en oeuvre.

```
<pick createInstance="no">
            <onMessage partnerLink="partner1"
                       portType="port1"
                       operation="operation1"
                       variable="var1">
                            ...
            </onMessage>
            <onMessage partnerLink="partner1"
                       portType="port1"
                       operation="operation2"
                       variable="var2">
                            ...
            </onMessage>
            <onMessage partnerLink="partner1"
                       portType="port1"
                       operation="operation3"
                       variable="var3">
                            ...
            </onMessage>
</pick>
```

- L'activité `scope` fournit un contexte pour une activité. Un `scope` contient une activité primaire, qui peut être structurée et qui définit son comportement normal. Un `scope` peut contenir des déclarations de variables (visibles seulement dans le `scope`), des définitions de `faultHandlers` qui représentent une gestion locale des erreurs ; il peut également inclure des `eventHandlers`.

CHAPITRE II : VÉRIFICATION DE MODÈLES

Des études ont montré que le coût de correction d'une erreur détectée lors de la phase des tests ou après que le logiciel ait été mis en production est de 5 à 10 fois plus élevé que si la même erreur avait été détectée dans une phase antérieure [38]. Dans certains cas, des bogues dans des systèmes critiques peuvent mettre la vie des gens en danger. Ces problèmes et erreurs sont souvent liés, non pas à des erreurs de codage, mais à des erreurs au niveau des spécifications ou de la conception. Pour ces raisons, il est intéressant de trouver un moyen qui permet de vérifier les spécifications d'un système avant de passer à la phase du développement. Cette vérification a pour objectif de garantir la validité du système en question et d'assurer l'absence de comportements non voulus.

Les méthodes formelles sont des techniques utilisant la logique mathématique pour raisonner sur des systèmes informatiques (matériels et logiciels) dans le but de prouver diverses propriétés de leur comportement, entre autres, leur exactitude par rapport à leurs spécifications. Dans l'utilisation d'une méthode formelle, il entre deux entités principales : la spécification et la vérification.

Une spécification formelle est basée sur un langage formel qui permet de décrire un comportement ou des propriétés, selon le cas ou l'approche, d'une façon abstraite sans entrer dans les détails. Elle présente les exigences que le système doit satisfaire d'une manière non ambiguë. En général, elle exprime formellement le résultat attendu d'un système. Quant à la vérification formelle, elle permet de s'assurer que le comportement du système satisfait réellement les propriétés décrites par la spécification.

La vérification de modèles est une catégorie de méthodes formelles qui utilise des notations formelles pour la spécification d'un modèle – qui représente une abstraction du système étudié – et pour la spécification des exigences comportementales, et les combinent avec une méthode de preuve efficace. Si les deux types de description ont été bien formulés cela va garantir que toutes les exigences qui peuvent être exprimées seront formellement

prouvables. Un modèle est obtenu par abstraction du système en fonction des propriétés à satisfaire. La vérification de modèles se base sur la vérification des propriétés par une exploration des états du modèle. La figure 2.1 illustre le processus général de la vérification de modèles ; le verdict, en cas d'erreur, va généralement présenter un contre-exemple, i.e. une instance du modèle qui ne satisfait pas les propriétés indiquées.

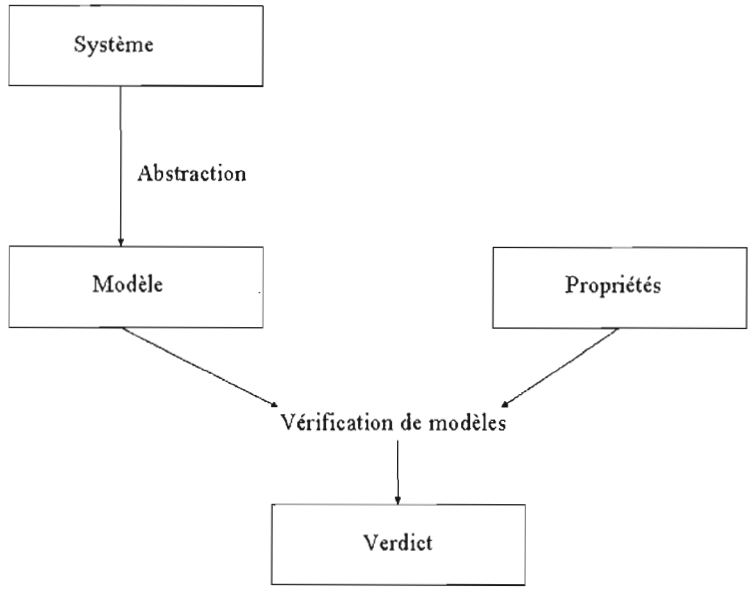


Figure 2.1 - Vérification de modèles

Ce chapitre présente d'un côté la vérification de modèles, son processus général et les types d'algorithmes utilisés. D'un autre côté, il expose avec plus de détails la notion de spécification de propriétés. Il explique également le type de propriétés que le logiciel que nous avons développé dans le cadre de notre recherche vérifie.

2.1 La vérification de modèles

Cette section introduit la vérification de modèles en décrivant son processus général et ses algorithmes.

2.1.1 Définition de la vérification de modèles

La vérification de modèles permet la vérification des systèmes. Pour ce faire, il faut utiliser des formalismes appropriés pour décrire ces systèmes et les propriétés que l'on souhaite vérifier.

Pour des raisons fondamentales, les techniques de vérification de modèles ne peuvent pas être appliquées aveuglement sur le système. Il faut d'abord commencer par créer un modèle simplifié qui sera appelé le modèle de vérification, où l'abstraction se fait en fonction des propriétés à vérifier. Par la suite, l'outil vérifie les propriétés sur le modèle obtenu. La vérification doit être exhaustive, c'est-à-dire l'outil doit explorer tout l'espace d'états pour s'assurer de la véracité de la propriété. Si la propriété est fausse, l'outil retourne un contre-exemple. Il faut alors s'assurer que ce contre-exemple peut se produire en réalité. Dans le cas où il n'est pas réalisable, il est possible que l'on ait omis une contrainte importante. Alors, on retourne au modèle pour en faire un autre plus sophistiqué. Si le contre-exemple peut se produire en réalité, ce cas est intéressant puisque nous avons détecté un problème au niveau du système.

2.1.2 Processus général de la vérification de modèles

Le processus de la vérification des modèles peut être résumé par les étapes suivantes [8]:

- Modélisation : consiste en la conversion du système à étudier en un modèle exprimé par un formalisme accepté par l'outil de vérification de modèles. Cette étape est souvent la plus compliquée parce qu'elle exige une bonne connaissance du domaine d'application ainsi que du langage de modélisation et elle nécessite l'utilisation de l'abstraction pour éliminer les détails.

- Spécification : avant de vérifier, il faut poser les propriétés qui doivent être satisfaites par le système étudié. On utilise souvent la logique temporelle qui permet d'affirmer comment le système évolue avec le temps.

- Vérification : elle est complètement automatisée ; dans le cas d'un résultat négatif une trace d'erreur est retournée : c'est ce que l'on appelle un contre-exemple, qui peut aider le concepteur à trouver la source de l'erreur.

2.1.3 Types d'algorithmes de vérification de modèles

Comme S. Merz [11] nous considérons deux types d'algorithmes :

- Algorithme global : ce type d'algorithme vise à identifier l'ensemble des états qui satisfont une propriété. Il se base sur la structure de la propriété à vérifier et évalue chacune de ses sous-formules sur l'ensemble du système étudié. La composante principale de cette approche est la procédure de marquage qui à partir d'une formule à vérifier φ va marquer pour chaque état q de l'automate représentant le système et pour chaque sous-formule ψ de φ, si ψ est satisfaite dans l'état q.

- Algorithme local : ce type d'algorithme vise à déterminer si un état donné satisfait une propriété. Il explore seulement certaines parties de l'espace d'états du système mais vérifie toutes les sous-formules de la propriété à vérifier. Dans ce type d'algorithme, le problème de vérification de modèles peut être vu comme étant de déterminer pour un système donné T et une propriété φ, est-ce qu'il existe une exécution de T qui ne satisfait pas φ ? Lorsque le système T et la propriété φ sont donnés par des automates, ceci peut se faire en regardant si le langage défini par l'intersection du langage pour T et du langage pour $\neg \varphi$ est vide.

Le problème majeur dont souffrent les algorithmes cités plus haut est l'explosion du nombre d'états. Pour remédier à ce problème, la vérification de modèle symbolique vient représenter de façon concise les états et les transitions du modèle et cela en utilisant différentes représentations. Les diagrammes de décisions binaires BDD (*Binary Decision Diagrams*) sont les plus utilisés, leur structure de données est simple à décrire et à réaliser. Ils

permettent de vérifier efficacement des systèmes de grande taille vu que les opérations de base sont peu onéreuses puisque la structure de données est compacte. Il y a également le BMC (*Bounded Model Checking*) qui est capable d'aborder des chemins très larges, vu qu'il ne considère que des séquences bornées et cherche un contre-exemple en examinant des chemins de plus en plus longs.

2.2 Spécification des propriétés

Pour énoncer formellement des propriétés comportementales des systèmes, on peut utiliser, entre autres, la logique modale ou la logique temporelle.

2.2.1 Logique modale

Une logique modale est une logique qui supporte les concepts de possibilité, d'existence et de nécessité. Une telle logique est utilisée pour exprimer des propriétés qui sont locales par rapport à l'état courant. Les opérateurs de base d'une telle logique sont :

- La possibilité : *<action> p* énonce qu'il est possible d'exécuter *action* et ensuite d'atteindre un état qui satisfait *p*.

- La nécessité : *[action] p* énonce que quand *action* se produit, l'état dans lequel on arrive satisfait nécessairement *p*.

On peut déduire d'autres concepts à partir de la négation de ceux cités en haut :

- L'impossibilité : *Non* <> qui est l'opposé de la possibilité.

- La contingence : *Non []* qui est l'opposé de la nécessité.

L'exemple ci-dessous (figure 2.2) montre un modèle (Modèle 1) où à partir de l'action « pièce », il est toujours possible de faire l'action « biscuits ». Par contre, cela n'est pas possible dans le cas du Modèle 2, à cause du choix non déterministe effectué au début.

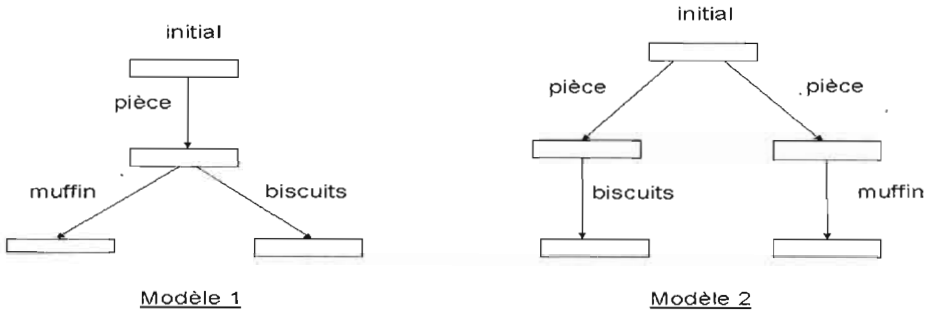


Figure 2.2 - Utilisation de la logique modale

En terme de traces (séquences d'actions), les deux modèles sont considérés comme ayant le même comportement puisque les deux génèrent l'ensemble suivant de traces : {pièce ; muffin, pièce ; biscuits}. Or, ces deux modèles n'ont pas nécessairement les mêmes propriétés modales, par exemple : [pièce] <muffin>True est vraie à l'état initial du Modèle 1, alors qu'elle ne l'est pas à l'état initial du Modèle 2.

Donc, la logique modale s'avère intéressante si on a besoin d'exprimer des propriétés locales [24].

2.2.2 Logique temporelle

Une logique temporelle est une forme de logique spécialisée dans les énoncés et raisonnements faisant intervenir la notion d'ordonnancement dans le temps [9]. Elle permet d'exprimer l'évolution de l'état d'un système. Ces propriétés peuvent être vues comme des comportements dynamiques. La logique temporelle dispose de connecteurs logiques (conjonction, disjonction, négation, implication et équivalence) et de connecteurs temporels qui se différencient d'une logique à une autre. Nous nous intéressons à deux types de logique temporelle [9] :

- Les logiques de temps linéaire (LTL, PLTL, ...) : elles permettent de spécifier des propriétés formulées sur des séquences d'états. Une telle logique n'est pas susceptible d'exprimer les exécutions alternatives qui sont générées aux instants où une sélection non déterministe est permise. Elle définit donc le comportement prévu du système en spécifiant l'unique futur possible (temps linéaire). Les combinateurs temporels usuels sont les suivants [9] :

  - *X p* : *p* est vraie à partir de l'état suivant (*Next*).
  - *F p* : *p* est vraie à partir d'un état futur ou de l'état actuel (*Finally*).
  - *G p* : *p* est vraie dans tous les états suivants, incluant l'état actuel (*Globally*).
  - *p1 U p2* : *p2* est vraie à partir d'un état *q* (futur ou actuel) et *p1* est vraie pour tous les états qui précédent l'état *q* (*Until*).

  On peut aussi combiner ces opérateurs, par exemple :

  - *GF p* : *p* est vérifiée une infinité de fois dans le futur.
  - *FG p* : *p* est toujours vérifiée à partir d'un certain état.

- La logique de temps arborescent CTL (*Computation Tree Logic*) : elle permet de spécifier des propriétés d'états. Ainsi, elle considère plusieurs futurs possibles à partir d'un état du système. Les connecteurs temporels sont *X*, *G*, *F* et *U* de la logique LTL, et des quantificateurs de chemin (combinateurs exprimant le côté arborescent) : *A p*, qui énonce que toutes les exécutions partant de l'état courant satisfont la propriété *p* ; *E p*, qui énonce qu'à partir de l'état courant, il existe une exécution satisfaisant *p*. Ces connecteurs s'utilisent par paire avec les autres, comme suit :

  - *EF p* énonce qu'il est possible d'atteindre un état où *p* est vérifiée.
  - *AF p* dit que pour toute exécution, il existe un état où *p* est vérifiée.
  - *AG p* dit que *p* est vraie partout (futur).

- *EG p* énonce qu'il existe une exécution où *p* est toujours vraie.
- *AX p* dit que tous les états immédiatement successeurs satisfont *p*.
- *EX p* énonce qu'il existe une exécution dont le prochain état satisfait *p*.

Le choix d'une logique dépend des spécificateurs de propriétés et des propriétés qu'on veut exprimer. Mais lorsqu'on utilise un outil de vérification de modèles, il faut respecter la logique qu'il utilise et spécifier les propriétés dans cette logique. Dans notre cas, nous avons utilisé l'outil *Spin/Promela* qui utilise entre autres la logique temporelle linéaire LTL, que nous expliquons brièvement même si notre propre outil n'utilise pas vraiment LTL, mais utilise plutôt les assertions de traces qui sont elles aussi disponibles dans *Spin*.

2.2.3 Procédure de la vérification de modèles LTL

La forme que doit avoir une exécution qui vérifie une propriété LTL φ peut être exprimée par une expression ω-régulière (expression utilisée pour manipuler des langages de mots infinis). La vérification de modèles LTL est fondée sur l'idée qu'on peut associer à chaque formule φ de LTL une expression ω-régulière qui décrit la forme que doit respecter une exécution si elle veut satisfaire φ. Du point de vue de la mise en oeuvre, il y a donc possibilité de représenter la vérification de modèles LTL par le biais de manipulation d'automates. Soit A l'automate représentant le comportement du modèle. Soit B l'automate représentant le comportement associé à la spécification (propriété LTL). Le modèle M va satisfaire la propriété φ si : $L(A) \subseteq L(B)$, donc si : $L(A) \cap \text{Compl}(L(B)) = \phi$. Pour vérifier si un modèle satisfait une propriété, on peut donc calculer $L(A) \cap \text{Compl}(L(B))$, et vérifier si le langage résultant est vide, comme l'illustre la figure 2.3.

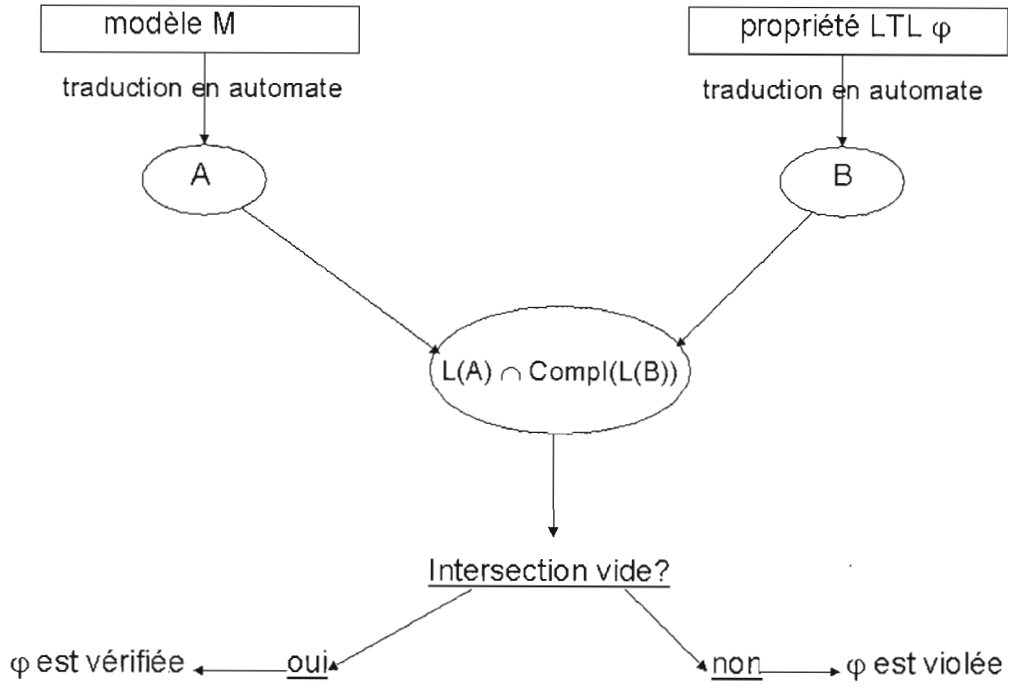


Figure 2.3 - Procédure de la vérification de modèles LTL

2.3 Les propriétés vérifiées par notre logiciel

Un processus *BPEL* peut avoir plusieurs comportements possibles. Plus précisément, un comportement peut être caractérisé par l'ensemble des messages échangés entre ce processus et son environnement (ensemble de ses partenaires). Ces comportements modélisent les choix possibles du processus à n'importe quel point de son exécution tel que perçu par un observateur externe (messages acceptés ou émis). Un processus satisfait une propriété si tous ses comportements la satisfont.

Pour notre logiciel de vérification de processus *BPEL* présenté au chapitre 5, on a décidé de vérifier l'ensemble des messages qu'un processus *BPEL* peut échanger avec son environnement. Donc, on veut surveiller l'ordre dans lequel ces messages sont échangés ainsi que leurs types pour voir s'ils se conforment à la spécification de l'interface du processus.

Cet ensemble de comportements représente les traces possibles (séquences de messages) du processus *BPEL* exprimées en termes de messages reçus ou émis. Il est possible en *Spin/Promela* de définir de telles traces avec des assertions de traces.

2.4 Spécifications des assertions de traces avec l'outil Spin

Une assertion de traces en *Spin* permet d'énoncer des propriétés par rapport aux canaux de messages (chapitre 4). Ceci se fait en définissant un automate de contrôle qui suit de près l'exécution du système. Cet automate ne change d'état que lorsqu'une opération d'envoi ou d'émission se trouvant à l'intérieur de la déclaration de l'assertion de traces est exécutée. Dans le cas où cette opération ne correspondrait pas à la transition de l'automate de l'assertion de traces, *Spin* va aussitôt signaler une erreur. Une assertion de traces ne peut contenir que des opérations d'envoi et de réception et elle doit toujours être déterministe [28].

Par exemple, une assertion de traces pour le processus `CreditFlow`, présenté dans le chapitre 4, est la suivante (en *Promela*, le souligné "_" représente une variable prédéfinie en écriture seulement, son rôle est de nous permettre de présenter une variable non définie dans un message lu sur un canal donné) :

```
trace {
    canal_client_In ? initiate,  input ;
    canal_creditRatingService_Out ! process,  input ;
    if
    ::canal_creditRatingService_In ? process,  output, _ ;
    ::canal_creditRatingService_In ? NegativeCredit,  _,fault;
    fi;
    canal_client_Out ! onResult,  output ;
 }
```

Cette assertion de traces montre que la réception du message (`initiate`, `input`) envoyé par `client`, va être suivie de l'envoi du message (`process`, `input`) au partenaire `creditRatingService` qui va répondre dans le cas normal par le message (`process`, `output`) ou par le message (`NegativeCredit`, `fault`) dans le cas d'erreur. Puis, le processus `CreditFlow` va répondre au `client` par le message (`onResult`, `output`).

Une assertion de traces doit respecter certaines propriétés :

- Les canaux de messages qui sont utilisés doivent être déclarés globalement.

- Les évènements d'envoi et de réception contenus dans les assertions de traces sont les seuls qui vont être surveillés et vérifiés.

- Une assertion de traces peut contenir, en plus des évènements d'envoi et de réception, des structures de contrôle comme `if`, `do`. Toutefois, cela exclut l'utilisation d'affectation ou de conditions. L'assertion de traces suivante montre que le processus étudié débute par la réception du message (`initiate`, `x`) du client. Puis, le processus entre dans une boucle dans laquelle il a le choix entre effectuer soit une opération d'incrémentation sur la variable `x`, soit une opération de décrémentation selon le message qu'il reçoit. Le message (`val`, `dummy`) permet de mettre fin à cette boucle. Finalement, le résultat est envoyé au client sous forme du message (`end`, `r`).

```
trace {
    canal_client_In ? initiate,  x, _ ;
    do
    ::canal_client_In ? inc,  x, _ ;
    ::canal_client_In ? dec,  x, _ ;
    ::canal_client_In ? val,  _, dummy ;  break ;
    od;
    canal_client_Out ! end,  r ;
}
```

- Une assertion de traces ne doit pas contenir de choix non déterministe. L'assertion de traces suivante n'est pas valide.

```
trace {
    if
    ::canal_seller1_In ? provide,  seller Data1 ;
      canal_seller2_In ? provide,  seller Data2 ;
      canal_seller3_In ? provide,  seller Data3 ;
    ::canal_seller1_In ? provide,  seller Data1 ;
      canal_seller3_In ? provide,  seller Data3 ;
      canal_seller2_In ? provide,  seller Data2 ;
    ::canal_seller2_In ? provide,  seller Data2 ;
      canal_seller3_In ? provide,  seller Data3 ;
      canal_seller1_In ? provide,  seller Data1 ;
    ::canal_seller2_In ? provide,  seller Data2 ;
      canal_seller1_In ? provide,  seller Data1 ;
      canal_seller3_In ? provide,  seller Data3 ;
    ::canal_seller3_In ? provide,  seller Data3 ;
      canal_seller1_In ? provide,  seller Data1 ;
      canal_seller2_In ? provide,  seller Data2 ;
    ::canal_seller3_In ? provide,  seller Data3 ;
      canal_seller2_In ? provide,  seller Data2 ;
      canal_seller1_In ? provide,  seller Data1 ;
    fi;
    ...
}
```

Cette assertion de traces signale que le processus étudié peut recevoir la suite de messages : (`provide, sellerData1`) du processus `seller1`, (`provide, seller2`) du processus `seller2` et (`provide, sellerData3`) du processus `seller3` et cela dans tous les ordres possibles comme il est montré dans l'assertion de traces. Ce phénomène apparaît dans les cas de processus qui contiennent des parties qui doivent s'exécuter d'une manière concurrente. Une telle assertion de traces contient un choix non déterministe comme on peut le remarquer – par exemple, les deux premières alternatives ont toutes deux l'évènement de réception du message (`provide, sellerData1`) sur `canal_seller1_In`. Or, le non déterminisme n'est pas supporté par les assertions de traces de *Spin*, ce qui nous a emmené à éliminer cette forme du non déterminisme, comme on va l'expliquer dans le chapitre 5. Néanmoins dans le cas présent le même comportement peut être exprimé par l'assertion de traces suivante :

```
trace {
    if
    ::canal_seller1_In ? provide,  sellerData1 ;
      if
      ::canal_seller2_In ? provide,  sellerData2 ;
        canal_seller3_In ? provide,  sellerData3 ;
      ::canal_seller3_In ? provide,  sellerData3 ;
        canal_seller2_In ? provide,  sellerData2 ;
      fi ;
    ::canal_seller2_In ? provide,  sellerData2 ;
      if
      ::canal_seller3_In ? provide,  sellerData3 ;
        canal_seller1_In ? provide,  sellerData1 ;
      ::canal_seller1_In ? provide,  sellerData1 ;
        canal_seller3_In ? provide,  sellerData3 ;
      fi ;
    ::canal_seller3_In ? provide,  sellerData3 ;
      if
      ::canal_seller1_In ? provide,  sellerData1 ;
        canal_seller2_In ? provide,  sellerData2 ;
      ::canal_seller2_In ? provide,  sellerData2 ;
        canal_seller1_In ? provide,  sellerData1 ;
      fi ;
    fi;
    ...
    }
```

CHAPITRE III : TRAVAUX CONNEXES

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. Ce chapitre présente certains de ceux que nous avons étudiés. Il présente également la notion de système distribué.

3.1 Vérification des systèmes distribués

Un système distribué est un ensemble d'entités s'exécutant de façon concurrente et qui partagent des ressources et communiquent entre elles, via des canaux ou des variables partagés, pour accomplir un traitement donné. Ces processus s'harmonisent de telle manière que le système est perçu de l'extérieur comme une seule unité.

La vérification du bon fonctionnement de ces systèmes devient une question capitale vu leur complexité et le coût notable des pannes. Ainsi, un système distribué peut ne pas fonctionner correctement même si chacune de ses parties isolées semble bien fonctionner. Ces types de pannes arrivent principalement au niveau des interactions entre les multiples exécutions concurrentes des composantes.

Les problèmes qui peuvent survenir dans un système distribué sont délicats à identifier avec les techniques standards de test, puisque plusieurs aspects de l'exécution d'un système distribué sont au-delà du contrôle du testeur, ce qui rend ardue la gestion des tests d'une façon reproductible [28]. Par contre, avec la vérification de modèles, on peut décrire et vérifier le comportement d'un système complexe à n'importe quel niveau d'abstraction.

3.2 Vérification du BPEL avec Spin/Promela

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

3.2.1 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 FIFO (*First In 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 [12].

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.

3.2.2 Travaux de Nakajima

Dans un premier temps, ce chercheur a vérifié avec *Spin* la composition de services Web exprimée avec WSFL [19]. 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 [18].

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 l'autre doit être fausse) [17].

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.

3.2.3 Travaux de Fernández, 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 le *BPEL* comme langage de définition de processus et les outils *Spin* et SMV (*Symbolic Model Verifier*) comme outils de vérification [21].

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.

3.3 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*.

3.3.1 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 [3]. 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 [3].

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.

3.3.2 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* [22]. 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 [23].

3.4 Synthèse

Les équipes présentées ci-haut ont opté pour la vérification de l'état interne du processus *BPEL* sans expliquer vraiment les types des propriétés qu'ils expriment – en d'autres mots les propriétés décrites en LTL portent sur l'état interne du processus, donc sur

des éléments possiblement non observables par un client/observateur externe. En ce qui nous concerne, nous avons vérifié si un processus satisfait son interface comportementale, qui représente son comportement externe (ordonnancement des messages qu'il envoie à ses partenaires ou reçoit d'eux), exprimée sous forme d'expression d'interface traduite en assertion de traces. De plus, dans la génération du modèle *Promela*, certaines équipes [2] passent par des représentations intermédiaires, alors que notre logiciel le fait directement.

CHAPITRE IV : OUTILS UTILISÉS

Ce chapitre traite des outils que nous avons utilisés pour mettre en œuvre notre application qui, à partir d'une spécification *BPEL* d'un processus et de son interface, permet de déterminer si le processus satisfait ou non l'interface, comme le montre la figure 4.1. Dans un premier temps, nous détaillerons l'outil *Spin* qui est le moteur utilisé pour mener cette vérification. Puis, nous détaillerons *Promela* qui est le langage de spécification utilisé par *Spin*. Ensuite, nous traiterons de la technologie DOM (*Document Object Model*) qui nous a été utile afin de faire la traduction d'une spécification *BPEL* en une spécification *Promela*, et nous terminerons par détailler l'API BPWS4J qui nous a permis d'extraire les éléments *BPEL* à partir de l'arbre fourni par cette API même.

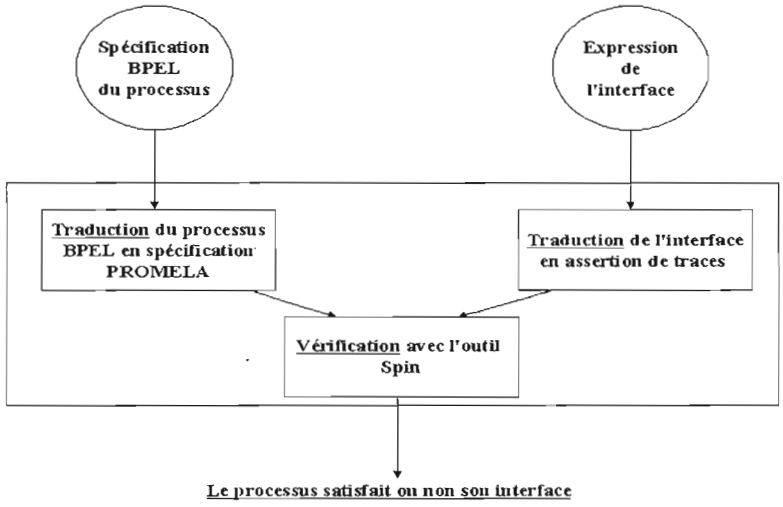


Figure 4.1 - Vue générale de notre logiciel de vérification

4.1 Spin/Promela

4.1.1 Spin

Spin (*Simple Promela INterpreter*) est un outil conçu pour la vérification des systèmes distribués. Son rôle est de montrer l'exactitude des interactions entre les processus constituant un système. Relativement à une spécification formelle du comportement du modèle (représentant une abstraction du système exprimée en *Promela*) et de diverses propriétés du comportement attendu de ce système (représentant les exigences de véracité) il vérifie si ces propriétés sont satisfaites.

L'outil *Spin* comporte en réalité deux modes :

- Un mode de simulation qui permet de simuler les comportements possibles du système.

- Un mode de vérification qui détermine si le modèle satisfait ou non une propriété LTL, un automate `never claim` ou une assertion de traces. Ceci est effectué en menant, si nécessaire, une vérification exhaustive c'est-à-dire en parcourant l'ensemble des états atteignables du système.

La stratégie de vérification de *Spin* est basée sur la recherche d'un contre-exemple (une exécution qui ne vérifie pas un comportement spécifié) [30]. À partir d'un modèle et d'une propriété, *Spin* détermine l'espace d'états de leurs automates correspondants. Pour trouver un contre-exemple, il applique l'algorithme de recherche en profondeur imbriquée (*Nested Depth-first Search)* sur cet espace d'états. Le but de cet algorithme est de trouver un cycle d'acceptation. Un cycle d'acceptation se résume en un état acceptable (état de terminaison pour l'automate) qui est atteignable de l'état initial du système et de lui-même. Si un tel cas existe, il en déduit que la propriété est violée. La particularité de cet algorithme est que le fait de trouver un cycle d'acceptation mettra fin à la recherche, donc il ne détecte pas tous les cycles d'acceptation contenus dans le graphe.

Comme de nombreux outils de vérification, *Spin* fait face au problème de l'explosion du nombre d'états. Pour diminuer son impact, il utilise la réduction d'ordre partiel qui est une

technique qui exploite l'indépendance des événements exécutés d'une manière concourante [31]. De cette façon, il réduit l'espace des états en sélectionnant seulement un sous-ensemble des chemins qui peuvent être entrelacés indépendamment.

Spin a aussi besoin de stocker de l'information sur les états. Pour réduire la mémoire nécessaire, il utilise une table de hachage; l'information est aussi stockée sous une forme compressée. Dans certains cas, même l'utilisation de ces techniques de réduction et de compression est insuffisante. *Spin* peut alors approcher le résultat de la vérification avec une exactitude qui dépend des ressources disponibles. Une des méthodes utilisées est le hachage du *bit-état* (*bit state hashing*) qui est une technique de compression avec perte. Cette technique peut empêcher *Spin* de détecter certains états d'erreurs mais il ne peut générer de faux contre-exemples [28].

4.1.2 Promela

Promela (*PROcess MEta LAnguage*) est un langage impératif qui ressemble au langage C mais enrichi de quelques primitives de communication. Une spécification *Promela* peut contenir trois types d'objets [29] :

- Les processus, qui sont des objets globaux. Un processus peut en activer d'autres à l'aide du mot clé `run`. Un modèle *Promela* peut contenir plusieurs processus mais il doit en contenir au moins un. Dans l'exemple ci-dessous, on définit un processus qui permet l'affichage d'un message (Hello world!) et on montre son initialisation au niveau de la partie `init`.

```
proctype processus_declaration(){
  printf("Hello world! \n")
}

init{
  run processus_declaration();
}
```

- Les variables, qui peuvent être de cinq types de données (`bit`, `bool`, `short`, `byte` et `int`) globales ou locales. *Promela* fournit aussi la possibilité de créer des

structures et des tableaux (d'une seule dimension). La définition ci-dessous déclare une structure contenant un champ entier et un autre booléen.

```
typedef type_structure {
      int var_int;
      bool var_bool;
}
```

- Les canaux de communication, qui sont utilisés pour le transfert de messages entre les processus. La communication via ces canaux est généralement asynchrone, mais on peut également définir un port rendez-vous pour la communication synchrone. Les champs d'un message ne peuvent pas être des tableaux. Une instruction d'envoi est exécutable si le canal n'est pas plein alors que celle de réception est exécutable si le canal n'est pas vide. Les messages sont insérés et retirés dans un ordre FIFO. Par exemple, la déclaration ci-dessous définit un canal pouvant contenir quatre éléments, entiers.

```
chan nom_channel = [4] of {int};
```

Promela offre également la possibilité de définir des séquences d'exécution atomiques (qui ne peuvent pas être interrompues) qui peuvent être déterministes ou non, des structures de sélection avec des gardes et des boucles, etc. L'exemple ci-dessous définit un `processus_moniteur` qui, de façon répétitive, à partir de la valeur du premier paramètre du message (`var1`), décide s'il va effectuer une opération d'addition ou de multiplication (sur `resultat`, avec `var2` le deuxième élément du message) et cela en invoquant d'autres processus. Le `timeout` assure que si aucun autre processus n'est actif et ne peut envoyer un message à `processus_moniteur`, alors son exécution se terminera.

```
chan transfert = [4] of {bool, int};
int resultat ;
resultat = 0 ;

proctype processus_moniteur(){
      do
      :: transfert? varl, var2 -> if
                                  :: varl == true ->
                                  run processus_add(var2);
                                  :: else          ->
                                  run processus_mul(var2);
                                  fi ;
      :: timeout               -> break ;
      od ;
}

proctype processus_add(int var){
      resultat = resultat + var ;
}

proctype processus_mul(int var){
      resultat = resultat * var ;
}
```

4.2 Technologie DOM

L'API DOM (*Document Object Model*) est une spécification W3C (*World Wide Web Consortium*) qui construit une structure arborescente d'un document XML. L'API DOM ne peut manipuler un document qu'en le chargeant intégralement en mémoire. Cela est requis dans certains cas de traitements XML. Cette API définit la structure logique du document sous forme d'une hiérarchie d'objets, comme le montre la figure 4.2. Chaque nœud de l'arbre possède un parent sauf la racine. On peut accéder à un élément en parcourant l'arbre ou en le référençant par son nom ou par son numéro, vu que les nœuds sont numérotés séquentiellement. De cette manière, n'importe quel élément de la structure arborescente peut être accédé à n'importe quel moment. DOM représente un standard indépendant de toute plateforme et de tout langage qui permet d'agir dynamiquement sur un document. Vu que DOM doit charger tout le document en mémoire pour le manipuler, il peut devenir désavantageux dans le cas de gros documents, puisque le temps de traitement devient long.

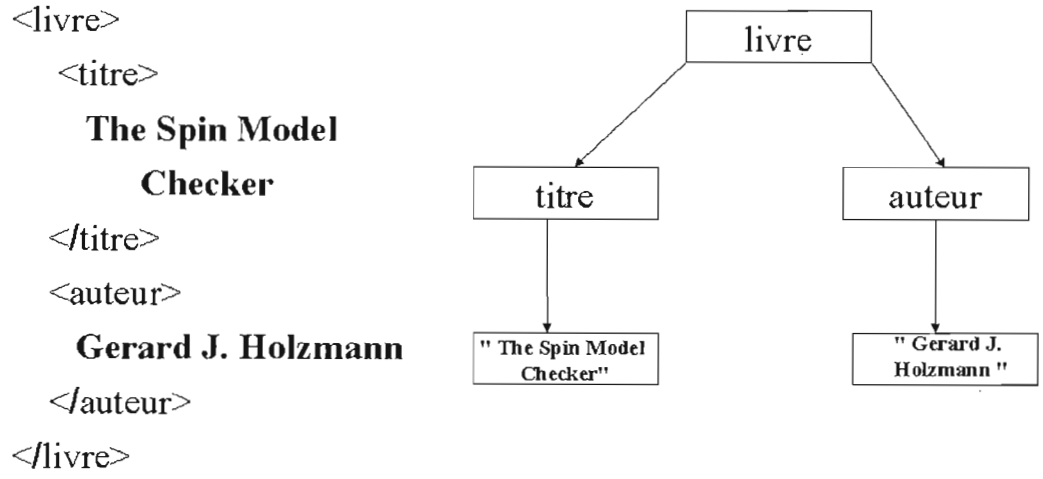
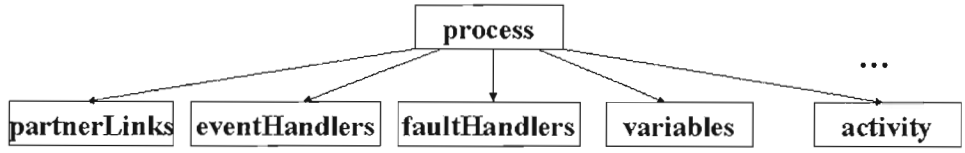


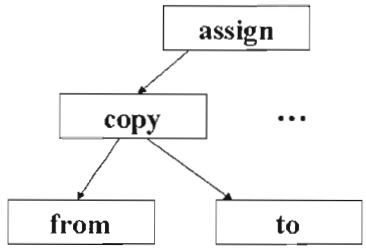
Figure 4.2 - Exemple basique d'arbre généré par DOM

4.3 BPWS4J: Business Process Web Service For Java

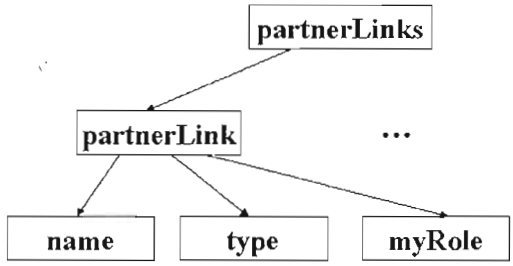
BPWS4J (*Business Process Execution Language for Web Services Java Runtime*) est une plate-forme qui fournit un moteur qui exécute des processus *BPEL* de la version 1.1. Cette mise en oeuvre fournie par IBM permet de créer et de modifier des fichiers *BPEL*. Elle comprend également un outil qui permet de valider syntaxiquement des fichiers *BPEL* en vérifiant si tous les attributs, exigés pour un élément par rapport à la spécification de la syntaxe de *BPEL* définie via la norme, ont été définis et si les entités qui ont été mises en référence sont réellement définies. BPWS4J offre une API intéressante qui permet d'accéder aux composantes d'un fichier *BPEL*. Cette API permet de créer la structure arborescente associée à un ficher *BPEL*. Chaque entité de l'arbre a ses propres attributs et peut être composée d'autres entités. L'entité racine est `BPWSProcess`, qui a comme attribut le nom du processus, et est constituée d'autres composantes comme le montre la figure 4.3. L'entité `Activity` peut être simple comme elle peut être complexe. On trouve plusieurs types d'`activity` comme cela a été expliqué au chapitre 1.



Composantes du BPWSProcess



Composantes de l'acitvité assign



Composantes de l'entité partnerLinks

Figure 4.3 - L'arbre généré par BPWS4J pour un fichier BPEL

Supposons qu'on a le fichier *BPEL* qui décrit le processus `CreditFlow` (il a été extrait du manuel d'instruction d'Oracle BPEL Process Manager (*Oracle BPEL Process Manager 2.0 Quick Start Tutorial*)) illustré à la figure 4.4. L'instruction `p = BPWSParser.readBPWSFromSource(is,"")` affecte à la variable `p`, de type `BPWSProcess`, la racine de l'arbre correspondant au processus *BPEL* décrit dans ce fichier, comme le montre la figure 4.5. La figure 4.6 représente les fils du nœud `scope` de la figure 4.5.

```
<process name="CreditFlow">
            <partnerLinks>
             .     <partnerLink name="client" />
                   <partnerLink name="creditRatingService" />
            </partnerLinks>
            <variables>
                   <variable name="input" />
                   <variable name="output" />
            </variables>
            <sequence>
                   <receive partnerLink="client"
                           operation="initiate" variable="input"/>

                   <scope>
                         <variables>
                               <variable name="crInput"/>
                               <variable name="crOutput"/>
                               <variable name="crFault"/>
                         </variables>
                         <sequence>
                          <assign>
                           <copy>
                                <from variable="input" part="payload"/>
                                <to variable="crInput" part="payload"/>
                           </copy>
                         </assign>
                         <invoke partnerLink="creditRatingService"
                               operation="process"
                               inputVariable="crInput"
                               outputVariable="crOutput">
                               <catch
                               faultName="nsxml0:NegativeCredit"
                               faultVariable="crFault">
                                  <assign>
                                     <copy>
                                        <from expression="20"/>
                                        <to variable="crOutput"
                                            part="payload"/>
                                      </copy>
                                  </assign>
                               </catch>
                         </invoke>
                         <assign>
                           <copy>
                            <from variable="crOutput" part="payload"/>
                            <to variable="output" part="payload"/>
                           </copy>
                         </assign>
                       </sequence>
            </scope>
            <invoke name="callbackClient" partnerLink="client"
                     operation="onResult" inputVariable="output"/>
      </sequence>
</process>
```

Figure 4.4 - Code BPEL du processus `CreditFlow`

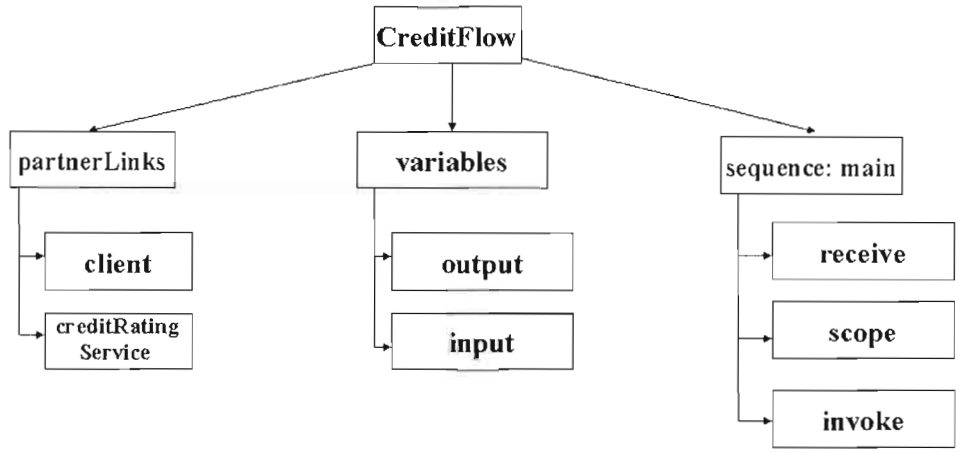


Figure 4.5 - Arbre résultant de l'API BPWS4J pour le processus `Creditflow`

Une fois que nous avons `p`, on extrait ses éléments comme suit, comme le montre la figure 4.5 en termes de structure BPWS4J :

- `p.getPartnerLinks()` retourne une variable de type `partnerLinks` qui contient la liste des `partnerLinks` du processus `CreditFlow` (`client`, `creditRatingService`).
- `p.getVariables()` renvoie la liste des variables du processus (`input`, `output`).
- `p.getActivity()` retourne la liste des activités du processus. Dans le cas du processus `CreditFlow`, on a deux activités basiques `receive` et `invoke`, et une activité composée `scope` qui à son tour contient divers éléments qu'on peut obtenir comme suit :

- `scope.getVariables()` retourne ses déclarations (les variables internes).

- `scope.getActivity()` retourne sa liste d'activités. Dans exemple, le `scope` contient une seule activité composée `sequence`. L'activité `sequence` est à son tour constituée des éléments suivants retournés par l'instruction `sequence.getActivities()`:

    - Deux activités `assign` dont les éléments `copy` sont retournés par l'instruction `assign.getCopies()`. Les éléments `from` et `to` sont retournés consécutivement par `copy.getFrom()` et `copy.getTo()`.
    - Un `invoke` qui contient un `catch` retourné par l'instruction `invoke.getCatchFaults()`.

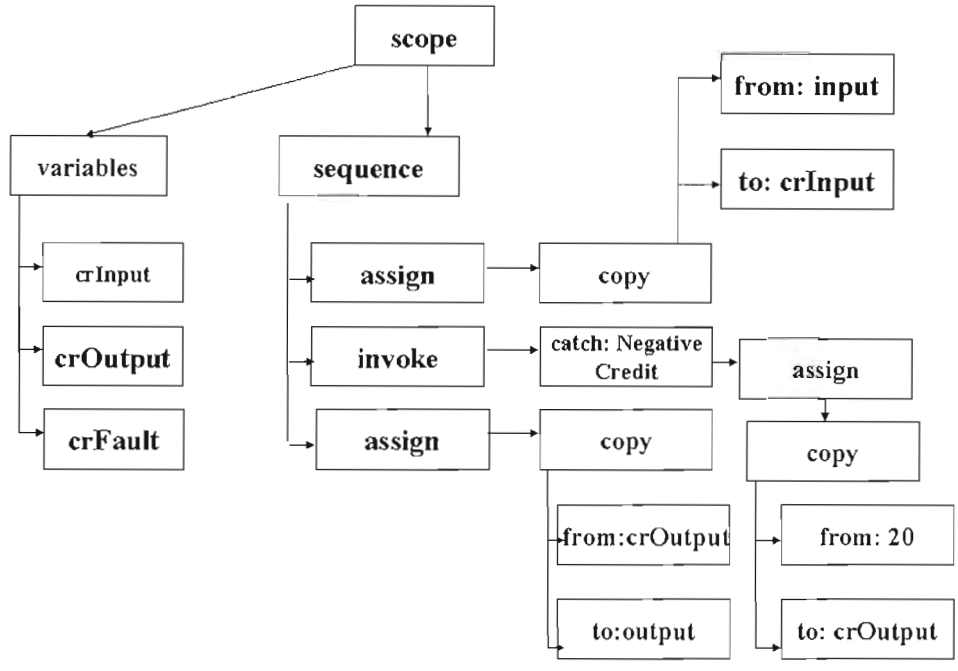


Figure 4.6 - Reste de l'arbre résultant de l'API BPWS4J pour le processus `CreditFlow` (Activité `scope`)

CHAPITRE V : SPÉCIFICATION ET MISE EN OEUVRE DES TRADUCTEURS

Ce chapitre présente la méthode que nous avons utilisée pour obtenir un modèle *Promela* et spécifier ses comportements possibles définis sous forme d'assertions de traces. Ce modèle et ces traces, qui sont obtenus à partir d'un processus *BPEL* et d'une spécification de son interface respectivement, ont été utilisés pour effectuer notre vérification avec *Spin*.

Au niveau du déroulement de notre travail, nous avons commencé par développer des traductions manuelles en nous assurant que nous fournissions une équivalence correcte du *BPEL* en *Promela*. Par la suite, le processus de traduction a été automatisé. La première section présente la procédure que nous avons suivie pour avoir un modèle *Promela* fermé à partir du processus *BPEL* initial. Pour la mise en oeuvre de cette procédure, nous avons utilisé l'outil BPWS4J et le langage Java. La deuxième section présente la procédure suivie pour obtenir les assertions de traces de ce modèle à partir d'expressions d'interface. Cette partie a été réalisée avec le langage Ruby.

5.1 Traduction de BPEL en Promela

5.1.1 Spécification du traducteur Promela

Notre logiciel effectue une abstraction du fichier *BPEL* pour avoir un modèle *Promela*. Nous n'avons pas considéré tous les éléments *BPEL* car nous nous intéressons au comportement normal du processus, donc nous avons exclu les `compensationHandlers`, les `faultHandlers` globaux et les `eventHandlers` locaux. Nous avons également limité notre travail en ne prenant pas en considération les `correlations`.

Nous avons traduit un processus *BPEL* en un processus *Promela* comme le montre le tableau ci-dessous. L'instanciation de ce processus se fait au niveau du bloc `init`. Son exécution est conditionnée (ceci est modélisé par le mot `provided`) par une variable booléenne `var_terminate`, c'est-à-dire, chaque fois que le processeur veut exécuter une instruction de ce processus il vérifie si cette variable est à `true`. Le processus se bloque

définitivement aussitôt que cette variable devient `false`. Nous avons ajouté cette variable dans le but de mettre fin à l'évolution du processus si une activité `terminate` est rencontrée (auquel cas la variable `var_terminate` est mise à `false`) au niveau du processus *BPEL*.

Syntaxe *BPEL*	Traduction *Promela* proposée

`<process name="nom" ...>`
`...`
`</process>`
	`proctype`
	`nom() provided(!var_terminate)`
	`{ ... }`

5.1.1.1 EventHandlers

Les `eventHandlers` sont utilisés pour représenter le cas d'un message asynchrone qui peut arriver n'importe quand pendant l'exécution du processus. Ce message donnera lieu à l'exécution d'une activité qui lui a été associée. Le processus est capable de recevoir ces messages simultanément à son exécution normale. Ces messages peuvent survenir un nombre arbitraire de fois tant que le processus est en exécution. Pour modéliser ce type d'éléments *BPEL* en *Promela*, chaque message (`onMessage`) d'`eventHandlers` est représenté comme un processus *Promela* actif (ce qui a été modélisé par le mot `active`), le corps de ce processus est une boucle (puisque le message peut survenir plusieurs fois) qui attend l'arrivée du message approprié pour exécuter l'activité correspondante, qui est simplement l'activité associée à ce message. Après un certain nombre d'exécution de la boucle qui dépend des fois où le message s'est produit, il sort de la boucle pour mettre fin à son exécution, comme le montre le tableau ci-dessous. Le rôle et fonctionnement des canaux sont expliqués dans le prochain paragraphe.

```
<eventHandlers> (globaux)
  <onMessage
     partnerLink = "partnerlink1"
     operation = "operation1"
     variable = "var1">
        activity1
  </onMessage>
  <onMessage
     partnerLink = "partnerlink2"
     operation = "operation2 "
     variable =  "var2">
        activity2
  </onMessage>
     ...
</eventHandlers>
```

```
active proctype
partnerlink1_operation1()
provided (!terminate_signal)
{
  do
  :: canal_partnerlink1_In ?
              operation1, var1
             -> activity1 ;
  :: true    -> break ;
  od ;
}

active proctype
partnerlink2_operation2()
provided (!terminate_signal)
{
  do
  :: canal_partnerlink2_In ?
              operation2, var2
           -> activity2 ;
  :: true  -> break ;
  od ;
}
...
```

5.1.1.2 PartnerLinks

Au niveau *Promela*, les messages échangés entre les processus sont constitués du nom de l'opération et du nom de la variable utilisée pour envoyer/recevoir le contenu du message. Les noms des opérations sont considérés comme étant des symboles déclarés à l'aide de `mtype` (énumération). Les `partnerLinks` sont vus comme des canaux utilisés pour échanger des messages (opérations plus variables) entre le processus et ses partenaires. Chaque `partnerLink` est représenté à l'aide de deux canaux : `canal_nomPartnerLink_In` pour les entrées (pour représenter la communication du partenaire vers le processus) et `canal_nomPartnerLink_Out` (pour représenter la communication du processus vers le partenaire) pour les sorties. De cette manière, le processus est lié à chacun de ses partenaires par deux canaux. Ces canaux sont déclarés globalement. Le tableau ci-dessous montre la traduction des `partnerLinks`.

<partnerLinks>
  <partnerLink name = "nom1"
  operation = "operation1" …>
  <partnerLink name = "nom2"
  operation = "operation2" …>
  …
</partnerLinks>
	mtype = {operation1, operation2,…};
	chan canal_nom1_In = [max] of
	{ mtype, type1, … } ;
	chan canal_nom1_Out = [max] of
	{ mtype, type2, … } ;
	chan canal_nom2_In = [max] of
	{ mtype, type3,… } ;
	chan canal_nom2_Out = [max] of
	{ mtype, type4,… } ;
	…

5.1.1.3 Variables

Les variables en *BPEL* sont dans la majorité des cas des structures de données (une variable peut avoir plusieurs champs). Dans notre modèle *Promela*, nous avons considéré les variables comme étant des structures de données déclarées à l'aide de `typedef` (cela va être expliqué par la suite). Le tableau ci-dessous montre la traduction de variables en *Promela*.

<variables>
  <variable
  name = "nom1"
  messageType = "messageType1"
  />
  <variable
  name = "nom2"
  messageType = "messageType2"
  />
  …
</variables>
	typedef type_messageType1 {
	  …
	}
	typedef type_messageType2 {
	  …
	}
	…
	type_messageType1 nom1, … ;
	type_messageType2 nom2, … ;
	…

5.1.1.4 Les activités

Les activités en *BPEL* sont catégorisées en deux groupes : celles de base et celles structurées.

Acivités de base :

Les activités de ce groupe peuvent être classées selon la tâche qu'elles effectuent :

a- Invocation d'opérations de services Web : l'activité `invoke` est utilisée par un processus *BPEL* pour appeler les services fournis par ses partenaires dans le but de réaliser une tâche donnée. Au niveau de l'abstraction de cette activité, nous avons gardé le nom du `partnerLink` sur lequel l'opération va s'effectuer, le nom de la variable qui va contenir le message de données et le nom de l'opération. Cet appel peut être :

- Synchrone (requête/réponse) : dans ce cas, l'appel exige une variable d'entrée et une autre de sortie. Cette forme d'`invoke` est traduite en *Promela* en un envoi du message (`operation, inputVar`) sur le canal de sortie du `partnerLink` (le canal sur lequel le processus envoie des messages à son partenaire `partnerLink` qui est `partnerLink_Out`) suivi d'une réception du message (`operation, outputVar`) sur le canal d'entrée du `partnerLink` (le canal sur lequel le processus reçoit des messages de son partenaire `partnerLink` qui est `partnerLink_In`), comme le montre le tableau ci-dessous.

`< invoke`
`partnerLink = "partnerLink1"`
`operation = "operation1"`
`inputVariable="inputVar"`
`outputVariable = "outputVar"`
`/>`
	`canal_partnerLink1_Out !`
	`operation1,inputVar;`
	`canal_partnerLink1_In ?`
	`operation1,outputVar;`

Dans le cas synchrone, l'opération peut retourner un message d'erreur. Cette erreur est caractérisée par un nom. La traduction de cette forme d'`invoke` se fait comme le cas précédent avec une différence au niveau du message reçu du `partnerLink` comme réponse qui peut être soit le message (`operation, outputVar`) dans le cas normal, soit un message d'erreur.

<invoke partnerLink = "partnerLink1" operation = "operation1" inputVariable = "inputVar" outputVariable = "outputVar" <catch faultName ="nomErreur" faultVariable = "variableName"> activity </catch> </invoke>	canal_partnerLink1_Out ! operation1, inputVar; if :: canal_partnerLink1_In ? operation1,outputVariable -> skip ; :: canal_partnerLink1_In ? nomErreur, variableName -> activity ; fi ;

- Asynchrone : dans ce cas, on utilise seulement une variable d'entrée puisqu'on n'attend pas de réponse. Cette forme d'`invoke` est traduite en *Promela* en un envoi du message (`operation, inputVar`) sur le canal de sortie du `partnerLink` concerné par l'opération, comme le montre le tableau ci-dessous.

< invoke partnerLink = "partnerLink operation = "operation" inputVariable = "inputVar" />	canal_partnerLink_Out ! operation, inputVar ;

b- Exécution d'opérations pour d'autres services Web : le processus fournit des services à ses partenaires par l'intermédiaire de l'activité `receive` et utilise l'activité `reply` pour envoyer la réponse correspondante. Au niveau de l'abstraction de ces deux activités, nous avons gardé le nom de l'opération, le nom du `partnerLink` et le nom de la variable. Le `receive` se traduit par une réception du message, composé du nom de l'opération et du nom de la variable utilisée pour recevoir la donnée, de la part du partenaire sur le canal d'entrée `partnerLink_In`. Le `reply` envoie une réponse à une demande faite auparavant par un `receive`. Nous l'avons traduit en *Promela* par l'envoi du message (`operation, variable_Output`) sur le canal de sortie qui correspond à ce partenaire, comme le montre le tableau ci-dessous.

<receive partnerLink = "partnerlink" operation = "operation" variable = "variable_Input" />	canal_partnerlink_In ? operation, variable_Input;
<reply partnerLink ="partnerlink" operation = "operation" variable = "variable_Output" />	canal_partnerlink_Out ! operation, variable_Output;

c- Mise à jour de variables : la mise à jour de variables en *BPEL* se fait par l'activité `assign`, qui peut être constituée de plusieurs `copy` (mises à jour de plusieurs variables). Ces variables peuvent être des structures de données, donc elles peuvent avoir des champs, auquel cas on peut avoir une copie entre variables ou entre champs ou encore entre champs et variables. Ces `copy` peuvent avoir plusieurs formes possibles comme le montre le tableau ci-dessous. Ces manipulations de variables sont traduites en *Promela* par des affectations.

<assign> <copy> <from = "var_source" part = "qname1"/> <to = "var_destination" part = "qname2"/> </copy>	var_destination.qname2 = var_source.qname1;
<copy> <from = "valeur"/> <to = "var_destination"/> </copy>	var_destination = valeur;
<copy> <from> literal_value <from/> <to = "var_destination"/> </copy>	var_destination = literal_value
<copy> <from expression = "general_expr" /> <to = "var_destination" property = "qname"/> </copy> </assign>	var_destination.qname = general_expr;

d- Terminaison d'un processus : vu que *Promela* ne dispose pas d'instruction *kill*, nous avons mis une variable `var_terminate` qui conditionne l'exécution des processus. Sa façon de fonctionner est expliquée un peu plus haut (la première section de ce chapitre) : la terminaison se traduit simplement par la mise à `true` de cette variable.

`<terminate/>`	`var_terminate = true ;`

Activités structurées :

Les activités structurées servent à décrire comment un processus *BPEL* est défini par la composition d'activités de base. Ces activités peuvent être catégorisées comme suit :

a- Activités d'ordre séquentiel : `sequence`, `switch` et `while`. L'activité `sequence` contient une ou plusieurs activités réalisées séquentiellement, dans l'ordre dans lequel elles sont listées dans la séquence. En *Promela*, cette activité a été traduite en un bloc contenant les activités de la séquence listées dans leur ordre initial séparées par des points virgules, comme le montre le tableau ci-dessous.

`<sequence>`
`  activity1`
`  activity2`
`  ...`
`</sequence>`
	`activity1 ;`
	`activity2 ;`
	`...`

L'activité `switch` permet d'exprimer le comportement conditionnel. Elle consiste en une liste ordonnée d'une ou de plusieurs branches avec conditions (`case`) suivies par une branche optionnelle `otherwise`. Ces branches sont considérées dans l'ordre de leur apparition. Si une branche `case` est vraie, l'activité qui lui a été associée est exécutée. En *Promela*, on peut définir un choix entre plusieurs exécutions avec la structure `if`. Si plus qu'une condition dans le `if` est vérifiée, la séquence correspondante est sélectionnée d'une

manière non déterministe. C'est pour cette raison que nous avons imbriqué les `if` en *Promela* pour mieux refléter la correspondance entre *BPEL* et *Promela* au niveau de la structure de sélection comme le montre le tableau ci-dessous.

<switch> <case condition = "condition1"> activity1 </case> <case condition = "condition2"> activity2 </case> <otherwise> activity3 </otherwise> </switch>	if :: condition1 -> activity1 ; :: else -> if :: condition2 -> activity2 ; :: else -> activity3 ; fi ; fi ;

L'activité `while` permet l'exécution répétitive d'une activité. Tant que la condition du `while` est vraie, l'activité est exécutée. Nous avons traduit cette activité avec le `do` du *Promela*. Quand la condition du `do` n'est plus vérifiée, la branche `else` du `do` est exécutée, dans laquelle nous mettons `break` qui permet d'achever l'exécution du `do`.

<while condition = "whileCondition"> activity </while>	do :: whileCondition -> activity; :: else -> break ; od;

b- Activité `pick` pour exprimer un choix non déterministe basé sur la réception d'évènements externes : l'activité `pick` est constituée de plusieurs branches évènement/activité. Une de ces branches va être sélectionnée lorsque l'évènement correspondant survient. Une activité `pick` doit contenir au moins un évènement `onMessage`. Cet évènement ressemble à l'activité `receive`, et c'est pour cette raison nous l'avons traduit de la même manière. Un `pick` a été traduit en *Promela* par un `if` avec diverses alternatives, dont l'une est `true` lorsqu'un évènement `onAlarm` est présent parmi les évènements du `pick` : la notion de temps n'existe pas en *Promela*, donc `true` représente ici simplement la possibilité quune alarme se déclenche.

```
<pick>
  <onMessage
      partnerLink ="partnerlink1"
      operation = "operation1"
      variable = "var1" >
               activity1
  </onMessage>
  <onMessage
     partnerLink ="partnerlink2"
     operation = "operation2"
     variable = "var2">
              activity2
  </onMessage>
  <onMessage
     partnerLink ="partnerlink3"
     operation = "operation3"
     variable = "var3">
              activity3
  </onMessage>
  <onAlarm (for=".." | until="..")>
            activity4
  </onAlarm>
</pick>
```

```
if
::canal_partnerlink1_In ?
       operation1,var1
       -> activity1;
::canal_partnerlink2_In ?
       operation2,var2
       -> activity2;
::canal_partnerlink3_In ?
       operation3,var3
       -> activity3;
::true -> activity4;
fi;
```

c- Activité de concurrence `flow` : les activités contenues dans un `flow` doivent être exécutées d'une manière concurrente. Un `flow` est traduit en *Promela* comme suit : nous avons défini pour chacune de ses activités un processus correspondant dont le corps contient l'activité en soi. Pour chacun de ces processus, nous avons déclaré une variable booléenne que nous initialisons au début à `false` et qui sera mise à `true` à la fin du processus correspondant pour signaler que le processus est terminé. Au niveau du processus principal, nous lançons l'exécution de ces processus. Puisqu'un `flow` ne se termine que lorsque toutes ses activités s'achèvent après le lancement de leur exécution, nous avons ajouté une condition pour la synchronisation qui bloque l'exécution du processus principal tant que tous les processus du `flow` ne sont pas terminés, c'est-à-dire, tant que la variable de terminaison de chacun des processus n'est pas mise à `true` comme il est illustré dans le tableau qui suit. Reste à signaler que les `links` utilisés par un `flow` pour exprimer les dépendances arbitraires de synchronisation n'ont pas été pris en considération par notre traitement de

`flow` – nous ne traitons donc que des `flows` qui lancent un ensemble de *threads*, sans synchronisation entre ces *threads* sauf à la fin du `flow` (barrière).

<flow>
activity1
activity2
activity3
</flow>
	bool process_1_termine = false;
	bool process_2_termine = false;
	bool process_3_termine = false;
	
	// Déclaration des processus
	proctype process_1()
	provided(!terminate_signal)
	{ activity1 ;
	process_1_termine = true;
	}
	proctype process_2()
	provided(!terminate_signal)
	{ activity2 ;
	process_2_termine = true;
	}
	proctype process_3()
	provided(!terminate_signal)
	{ activity3 ;
	process_3_termine = true;
	}
	
	// Activation des processus
	run process_1() ;
	run process_2() ;
	run process_3() ;
	
	// Barrière pour la synchronisation
	process_1_termine && process_2_termine &&
	process_3_termine ;

d- Activité `scope` : nous avons considéré dans la traduction de l'activité `scope` la déclaration des variables et les activités. Ceci a été traduit en *Promela* par une déclaration locale de ces variables suivie des activités incluses dans le `scope`, comme le montre le tableau ci-dessous.

<scope > <variables> … </variables> activity </scope>	variables ; activity ;

5.1.2 Mise en œuvre

Pour récupérer les éléments d'un processus *BPEL*, nous avons utilisé l'API BPWS4J vu dans le chapitre 4. Avec la commande `p = BPWSParser.readBPWSFromSource(is,"")`, on a `p` qui pointe vers la racine de l'arbre syntaxique qui représente le processus *BPEL*.

L'arbre représentant le processus *BPEL* va être parcouru à l'aide de cette API. Dans un premier niveau du parcours nous extrayons quatre éléments principaux du processus (que nous avons gardé pour le modèle *Promela*) qui sont:

- Les partenaires : `pls = p.getPartnerLinks()`.
- Les variables : `vars = p.getVariables()`.
- Les évènements : `e = p.getEventHandlers()`.
- L'activité du processus : `a = p.getActivity()`.

5.1.2.1 PartnerLinks

L'ensemble des noms des opérations effectuées par le processus et ses partenaires est sauvegardé dans une structure de type ensemble (`tableOperations`) qui est mise à jour chaque fois qu'on rencontre une activité de type `receive`, `reply`, `invoke` et `onMessage`. Ces opérations sont considérées comme des noms symboliques que nous avons introduits par une déclaration `mtype`.

Dans la déclaration des canaux, on doit connaître tous les types de variables transférés sur un canal donné. Or, dans le cas où le WSDL approprié n'est pas disponible, on ne peut le savoir qu'en parcourant l'ensemble des activités de type suivant : `receive`, `reply`, `invoke` et `onMessage`. Alors, si on était certain d'avoir le WSDL correspondant, la méthode que nous avons proposée ne serait pas vraiment nécessaire: Mais, comme nous n'avons pas pu obtenir ces fichiers WSDL pour les divers exemples, notre méthode reste utile.

Cette méthode consiste à parcourir dans un premier temps l'arbre pour extraire les `partnerLinks` et les variables et explorer ensuite l'ensemble des activités constituant le processus pour déterminer les types de variables transférées sur chaque `partnerLink`. Un `partnerLink` (`partnerLink_In` pour les entrées et `partnerLink_Out` pour les sorties) est sauvegardé dans une structure `tablePartnerLinks` qui est un `Map` dont la clé correspond au nom du `partnerLink` et la valeur à l'ensemble des types de variables qu'on peut transmettre sur ce `partnerLink`. Nous faisons la déclaration des canaux à partir de l'information mémorisée dans cette structure.

5.1.2.2 Variables

Les variables sont sauvegardées dans une structure `tableVariables` qui est un `Map` dont la clé est le nom de la variable et la valeur est le type de la variable. On ne peut connaître la nature des types utilisés que par l'intermédiaire des opérations dans le cas où le WSDL approprié n'est pas disponible (comme nous l'avons déjà écrit dans la section 5.1.2.1). Pour ce faire, nous avons proposé une analyse (inférence) qui sera expliquée par la suite.

Dans la majorité des cas, ces types sont des structures de données, mais cela n'est détecté que lors des parcours des activités de type `assign` ou bien au niveau des expressions arithmétiques ou booléennes. Notre programme déclare un `Map tableMessageTypes` dont la clé est le type et la valeur est un ensemble de `part` ou de `property` qui représente les champs de la structure qui est mise à jour chaque fois qu'on détecte un nouveau champ utilisé par ce type. Reste le cas suivant à signaler : si après le parcours de l'arbre aucun champ n'a été trouvé pour un type donné alors ; pour que notre représentation reste acceptable, nous ajoutons un champ par défaut dont le type est une chaîne de caractères – de cette manière, un type de structure de données qui ne dispose d'aucun champ aura au moins un champ par défaut.

Une structure `typedef` qui représente le type peut avoir un ou plusieurs champs. Les types de ces champs sont déduits à partir des traitements effectués sur eux et cela au fur et à mesure qu'on exploite les activités du processus traité. Supposons qu'on a la déclaration de variable *BPEL* suivante:

```
<variable name="shipRequest"
          messageType="sns:shippingRequestMsg"/>
```

Après le parcours du processus *BPEL*, cette déclaration pourrait par exemple générer le code *Promela* suivant :

```
typedef type_shippingRequestMsg {
  int  itemsTotal;
  int  payload;
  type_chaineCaracteres itemsCount;
  bool  shipComplete;
}
```

Les champs mis dans la déclaration de la structure sont détectés lors du parcours de l'arbre syntaxique correspondant au processus *BPEL*, c'est-à-dire lors du traitement des activités `assign` et des expressions booléennes et arithmétiques. Les types de champs sont choisis de la manière suivante :

1- *nomvar.champ = nomvar1.champ1 +/- nomvar2.champ2*

Cette instruction en *BPEL* veut dire implicitement que les champs *champ*, *champ1* et *champ2* ne sont pas des chaînes de caractères mais plutôt des types numériques – la spécification du langage *BPEL* indique que les seules opérations permises sur des chaînes de caractères sont = et !=. Nous avons choisi le type `int` pour représenter les types numériques en *Promela*.

2- *nomvar1.champ1 </>/<=/>= nomvar2.champ2*

Même traitement que celui indiqué plus haut.

3- *nomvar.champ =/==/!= true /false* (dans une condition `case` ou `while`)

On en déduit que *champ* ne peut être qu'une variable booléenne. On a choisi le type `bool` en *Promela* pour représenter ces variables.

4- *nomvar.champ = expression*

Si *expression* est un entier (définition de la fonction `isDigit`) alors *champ* est de type `int` sinon on en déduit qu'il est de type chaîne de caractères. Pour le type chaîne de caractères, nous avons déclaré en *Promela* un type `type_chaineCaracteres` pour le représenter. Nous avons défini également la chaîne vide.

5- *nomvar1.champ1 = nomvar2.champ2*

Les *champ1* et *champ2* doivent être du même type. Si les deux champs (ou un des deux) ont été déjà déclarés, on teste s'ils sont du même type, sinon on modifie le champ d'un des deux.

Pour les variables définies avec `types` (non pas avec `messageTypes`) nous avons fait le même traitement.

Reste un cas particulier à signaler. Supposons qu'on trouve dans un fichier *BPEL* :

```
1) <assign>
     <copy>
       <from variable="shipRequest" property="props:itemsCount"/>
       <to variable="shipNotice" property="props:itemsCount"/>
     </copy>
   </assign>
```

Puis :

```
2) <assign>
      <copy>
         <from expression=
                    "bpws:getVariableData('itemsShipped')
                     +
                     bpws:getVariableProperty('shipNotice',
                     'props:itemsCount')"/>
         <to variable="itemsShipped"/>
      </copy>
   </assign>
```

Avec notre traducteur *Promela*, on aura :

```
shipNotice.itemsCount = shipRequest.itemsCount ;
```

Puis:

```
itemsShipped = itemsShipped + shipNotice.itemsCount ;
```

Arrivé au premier `assign`, les deux champs des deux structures ont comme type le type par défaut car jusqu'à présent aucune information ne dit le contraire. Mais, arrivé au deuxième `assign`, on conclut que le champ de la variable `shipNotice.itemsCount` est un `int`. Pour régler ce problème, nous établissons une relation de causalité entre les champs ; de cette manière si le type d'un champ est mis à jour, tous les champs en relation avec (relation de causalité) vont être mis également à jour. Et la déclaration :

```
typedef type_shippingRequestMsg {
  int  itemsTotal;
  int  payload;
  type_chaineCaracteres  itemsCount;
  bool  shipComplete;
}
```

devient alors :

```
typedef type_shippingRequestMsg {
  int  itemsTotal;
  int  payload;
  int  itemsCount;
  bool  shipComplete;
}
```

5.1.2.3 Traitement des expressions

Pour les expressions booléennes qu'on trouve au niveau des conditions des `switch` et `while`, il y a plusieurs cas possibles qu'on a essayé de généraliser de la manière suivante :

- `ExpressionBooléenne = not? (opérande op opérande) | opérande = (true ou false) | opérande`
- `Opérande = bpws:getVariableProperty('nomvar','props:nomProperty'?) | bpws:getVariableData('nomvar','nomPart'?)`
- `op = < | > | <= | >= | = | !=`
- `not = négation` (dans ce cas on inverse l'expression)

Le traitement d'expressions arithmétiques qu'on trouve dans des activités `assign` est le même que celui d'expressions booléennes à deux différences :

- `ExpressionArithmétique = opérande op opérande | opérande`
- `op = + | - | * | /`

5.1.2.4 Traitement des activités

Pour le traitement des activités, comme on l'a déjà signalé, l'instruction `a = p.getActivity()` retourne l'activité principale du processus *BPEL* qui contient un ensemble d'activités imbriquées ou de même niveau. L'extraction de l'ensemble de ces activités est faite à l'aide d'appels récursifs. On peut différencier trois types de traitement des activités :

- Le traitement des activités atomiques consiste à retourner l'information qu'elles contiennent sans faire d'appel récursif – c'est le cas pour `receive`, `reply`, `terminate` et `wait`.

- Le traitement des activités composées comme `sequence`, `scope`, `switch`, `while`, `pick` et `flow` consiste à faire d'autres appels récursifs jusqu'à ce qu'on arrive au cas de base (le fait d'arriver à une activité atomique).

- Finalement, le traitement d'un troisième type qui ressemble aux opérations atomiques et qu'on va appeler des activités semi-composées. Ce traitement fait des appels à d'autres fonctions pour extraire certaines informations, par exemple les `assign` (traitement des `to` et `from`), `invoke` (traitement des `faultHandlers`).

5.1.3 Fermeture du modèle

Le modèle *Promela* obtenu à partir d'un processus *BPEL* est ouvert : un système ouvert est un système qui se caractérise par le fait qu'il interagit avec son environnement.

Certains outils de vérification permettent la vérification de modèles ouverts cela en simulant leurs environnements. Sauf que *Spin* n'accepte que des modèles fermés – c'est-à-dire, le modèle doit contenir toutes les informations qui sont exigées pour établir la vérification [28]. Un modèle fermé inclut toutes les entrées et les paramètres de l'environnement, ce qui oblige à encapsuler dans le modèle *Promela* toutes les hypothèses (les entrées et évènements qui affectent l'exécution du modèle) concernant les processus externes avec lesquels le processus en question interagit.

Le processus *BPEL* modélisé peut interagir avec un ou plusieurs autres processus qui sont présentés dans les `partnerLinks`. L'ensemble de ces processus constitue l'environnement du processus étudié. Pour fermer le modèle, il faut modéliser également ces processus partenaires en *Promela*. Alors, nous les avons présentés comme étant des processus indépendants qui envoient des messages au processus étudié et en reçoivent. Donc, le corps de chacun de ces processus représente des envois de messages que le processus etudié est capable de consommer en prenant en considération tous les choix possibles.

La figure 5.1 représente un des processus qu'on a étudié qui est le processus `purchaseOrder` [1]. `Purchasing`, `Invoicing`, `Shipping` et `Scheduling`. Le processus `Purchasing` joue le rôle du client qui active le processus `purchaseOrder` par l'envoi d'une commande et puis attend une réponse. Une fois que le processus `purchaseOrder` reçoit cet ordre, il entre en communication avec les autres partenaires pour accomplir sa tâche et répondre au client. Les flèches avec les noms représentent les différents messages de l'ensemble de la communication. Chacun des partenaires doit savoir à quel moment il doit envoyer tel message ou en consommer. Et chacun d'eux réagit en fonction de l'état interne du processus `purchaseOrder`.

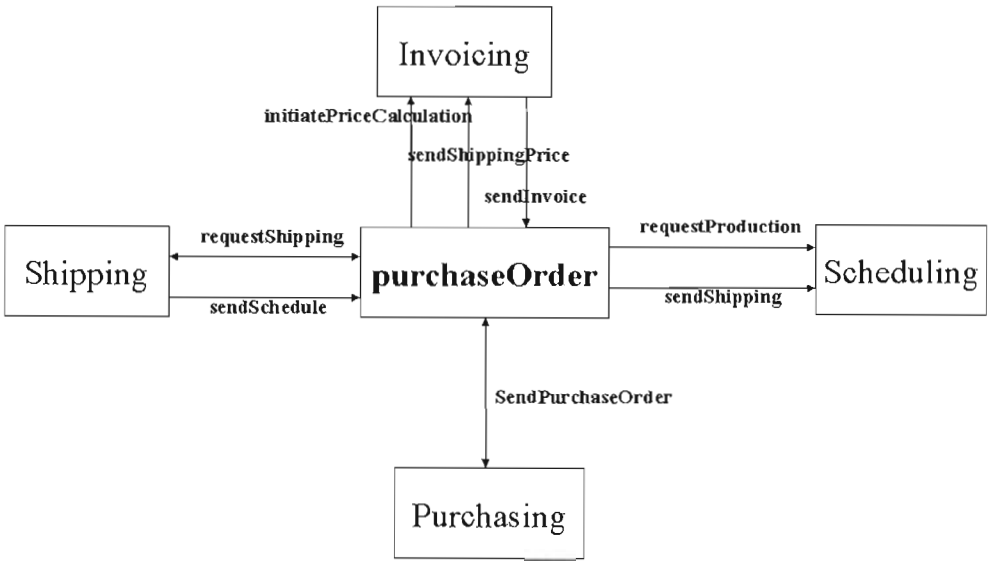


Figure 5.1 - Représentation graphique du processus purchaseOrder

À partir de cet exemple et d'autres étudiés, la génération de l'environnement (ensemble des partenaires) d'une manière automatisée s'est avérée délicate, vu que cet environnement doit, d'une certaine façon, avoir des informations sur l'état interne (implicite) du processus en question [3]. Nous avons donc plutôt supposé que les spécifications des différents partenaires sont effectivement fournies en *BPEL* – lorsque ce n'est pas le cas, nous avons nous-même développé ces modèles. Donc, on peut générer leurs modèles *Promela* avec notre logiciel.

5.2 Traduction des spécifications d'interfaces en assertions de traces

5.2.1 Spécification du traducteur d'assertions de traces

La vérification que nous voulons effectuer consiste à s'assurer que le processus *BPEL* se comporte correctement, et ce relativement à la description de son interface externe. Nous avons opté pour une interface qui représente le modèle comportemental du processus, mais

d'une façon abstraite. Cette interface capture les interactions dans lesquelles ce processus s'engage pour effectuer une tâche précise. Les activités décrites dans une interface correspondent aux messages envoyés et reçus par le processus. Cette interface décrit les types de messages, les directions dans lesquelles ces messages sont échangés, ainsi que les dépendances entre eux. Cette interface ne décrit pas les tâches internes comme la manipulation des données. Toutefois, elle se concentre sur un seul participant, qui est le processus qu'on veut analyser, les autres étant vus comme des partenaires.

Techniquement, nous avons représenté cette interface par une expression d'interface qui décrit le comportement attendu du processus sous forme de séquences possibles de messages envoyés au/reçus par le processus. Pour décrire ces expressions d'interfaces, nous avons utilisé la notation suivante qui ressemble aux expressions régulières:

- ?m réception d'un message m.
- !m envoi d'un message m.
- !?m envoi d'une requête m suivie immédiatement par la réception de la réponse.
- e1 [] e2 = choix entre e1 et e2.
- e1 || e2 = exécution concurrente (entrelacement) de e1 et e2.
- {e} = 0 ou 1 occurrence de l'expression e.
- e+ = 1 ou plusieurs occurrences de l'expression e.
- e* = 0, 1 ou plusieurs occurrences de l'expression e.
- (e) = pour indiquer une sous expression et indiquer explicitement la priorité.

Pour chacune des expressions ci-dessus, nous avons proposé une traduction adéquate en assertions de traces – voir section 5.2.2 pour la description de ces assertions dans le contexte de *Spin*.

Prenons deux exemples pour présenter la forme de ces expressions d'interface :

1- L'expression d'interface du processus `CreditFlow` – présenté au chapitre 4 – est la suivante :

```
canal_client_In ? initiate,  input ;
canal_creditRatingService_Out ! process,  input ;
( canal_creditRatingService_In ? process,  output, _
  []
  canal_creditRatingService_In ? NegativeCredit, _, fault
);
canal_client_Out ! onResult,  output
```

Cette expression d'interface veut dire que le processus `CreditFlow` reçoit tout d'abord un message (`initiate,input`) du partenaire `client`. Par la suite, il envoie le message (`process,input`) à son partenaire `creditRatingService`, qui à son tour répond soit par le message (`process,output,_`), soit par le message (`NegativeCredit,_,fault`). Finalement, le processus répond au `client` par le message (`onResult,output`). Pour que nous puissions dire que le processus `CreditFlow` se comporte correctement, l'ensemble de ses exécutions possibles doit respecter l'ordre et les types des messages décrits par son expression d'interface.

L'assertion de traces de *Spin* correspondant à cette expression d'interface est la suivante :

```
trace {
  canal_client_In ? initiate,  input ;
  canal_creditRatingService_Out ! process,  input ;
  if
  ::canal_creditRatingService_In ? process,  output, _ ;
  ::canal_creditRatingService_In ? NegativeCredit,_, fault;
  fi;
  canal_client_Out ! onResult,  output ;
 }
```

Cette trace décrit des exigences sur le comportement du processus `CreditFlow`, cela en définissant l'ordre des messages, les types de messages et en spécifiant les canaux utilisés.

2- L'expression d'interface du processus `auctionService` [1] – décrit dans l'annexe B – est la suivante :

```
( canal_seller_In ? provide,  sellerData
||
  canal_buyer_In ? provide,  buyerData
);
 canal_auctionRegistrationService_Out ! process,
                              auctionRegistrationData;
 canal_auctionRegistrationService_In ? answer,
                    auctionRegistrationResponse;
( canal_seller_Out ! answer,  sellerAnswerData
||
  canal_buyer_Out ! answer,  buyerAnswerData
)
```

Cette expression d'interface dit que le processus `auctionService` reçoit concurremment les messages (`provide,sellerData`) et (`provide,buyerData`) de ses partenaires `seller` et `buyer` respectivement. Par la suite, il envoie le message (`process,auctionRegistrationData`) au processus `auctionRegistrationService` qui à son tour répond par le message (`answer,auctionRegistrationResponse`). Le processus envoie les messages (`answer,sellerAnswerData`) et (`answer,buyerAnswerData`) aux processus `seller` et `buyer` respectivement comme réponses à leurs requêtes initiales. Pour que nous puissions conclure que le processus `auctionService` respecte son expression d'interface, il faut que l'ensemble de ses séquences d'exécution respecte l'ordre des messages décrit par cette expression d'interface, par exemple, la présence du message (`process,auctionRegistrationData`) sur `canal_auctionRegistrationService_Out` doit être suivie par le message (`answer,auctionRegistrationResponse`) sur le canal `canal_auctionRegistrationResponse_In` et ainsi de suite.

L'assertion de traces de *Spin* correspondant à cette expression d'interface est la suivante :

```
trace {
  if
  ::canal_seller_In ? provide,  sellerData ;
    canal_buyer_In ? provide,  buyerData ;
  ::canal_buyer_In ? provide,  buyerData ;
    canal_seller_In ? provide,  sellerData ;
  fi;
  canal_auctionRegistrationService_Out ! process,
                                  auctionRegistrationData ;
  canal_auctionRegistrationService_In ? answer,
                              auctionRegistrationResponse ;
  if
  ::canal_seller_Out ! answer,  sellerAnswerData ;
    canal_buyer_Out ! answer,  buyerAnswerData ;
  ::canal_buyer_Out ! answer,  buyerAnswerData ;
    canal_seller_Out ! answer,  sellerAnswerData ;
  fi;
 }
```

Cette assertion de traces décrit des exigences sur le comportement du processus `auctionService`. Elle ressemble à la trace du processus `CreditFlow` à une différence : la présence de la concurrence, que nous avons représentée par le premier et le dernier `if`. La traduction de la concurrence au niveau des assertions de traces va être expliquée par la suite.

5.2.2 Mise en œuvre du traducteur d'assertions de traces

Cette partie du logiciel a été programmée en langage Ruby [35] et à l'aide d'une approche orientée-objet. Nous avons défini une classe abstraite de base `InterfaceExpr` et cinq sous-classes qui en héritent (figure 5.2). `InterfaceExpr` contient les opérations (comme l'indentation) de base communes aux sous-classes. Chacune de ces sous-classes contient un ensemble de fonctions dont les deux principales sont `to_s` et `to_trace` : `to_s` génère une représentation sous forme de chaîne de caractères à partir de l'objet correspondant, alors que `to_trace` génère, à partir de l'expression d'interface (syntaxe décrite au début de 5.2.1), une assertion de traces. Dans ce qui suit nous nous restreindrons à la description de `to_trace`.

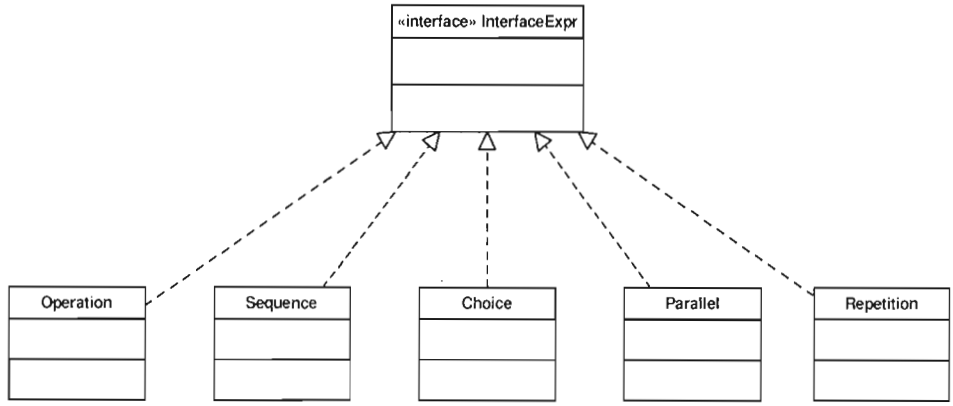


Figure 5.2 - Diagramme de classe du traducteur d'assertions de traces

5.2.2.1 Opération

Pour une opération simple, la fonction `to_trace` retourne simplement une chaîne de caractères représentant l'opération correspondante. Par exemple, soit:

```
Operation.new(:NegativeCredit,:In,:creditRatingService,:_, :fault)
```

Ceci est un appel du constructeur de la sous-classe `Operation` pour créer un objet dont le champ `partner` est `creditRatingService`, le champ mode est `In`, le message est `NegativeCredit`, suivi de la liste des variables de cette opération. Pour faciliter l'écriture des expressions d'interface, nous avons utilisé des constructeurs auxiliaires. Pour cette sous-classe, nous avons catégorisé ces constructeurs selon le mode d'opération : `In`, `Out`, etc.

Pour la fonction `to_trace`, la chaîne retournée dans le cas de l'opération citée ci-haut serait la suivante :

```
canal_creditRatingService_In ? NegativeCredit, _, fault;
```

5.2.2.2 Séquence

Le constructeur de la sous-classe `Sequence` reçoit comme entrée une variable de type tableau. La fonction `to_trace` parcourt le tableau élément par élément. Pour chaque

élément, elle appelle sa fonction `to_trace`. De cette manière, chaque type d'élément est généré tel qu'il est décrit dans sa classe.

5.2.2.3 Choix

Le constructeur de la sous-classe `Choice` reçoit une liste d'éléments. La fonction `to_trace` fonctionne comme dans le cas d'une séquence à la différence suivante : les éléments sont générés comme étant des branches d'un `if`. Chaque branche représente un cas à explorer possiblement. Soit l'exemple suivant :

```
choice(_in(:creditRatingService,:process,:output,:_),
       _in(:creditRatingService,:NegativeCredit,:_,:fault)
      )
```

La conversion en assertion de traces donne:

```
if
::canal_creditRatingService_In ? process,  output, _ ;
::canal_creditRatingService_In ? NegativeCredit,  _,fault ;
fi;
```

5.2.2.4 Parallèle

Le constructeur de la sous-classe `Parallel` reçoit une liste d'éléments. La conversion du parallélisme en assertion de traces nécessite l'élimination de la concurrence puisqu'une assertion de traces ne peut contenir que des `sequences`, `if` et `do`. Nous avons donc décidé de générer toutes les combinaisons possibles pour des cas généraux de concurrence. Pour ce faire, nous nous sommes basés sur les règles suivantes (signalons que lorsque nous mettons une lettre majuscule cela veut dire qu'elle est une activité composée qui peut être dans notre cas une séquence, un choix, etc. et que lorsque nous mettons une lettre minuscule cela veut dire qu'il s'agit d'une simple opération d'envoi ou de réception):

- (a || b) = (a; b) [] (b; a)
- A || B || C

  Avec:

  A = a1; a2; a3

  B = b1; b2; b3

  C = c1; c2; c3

Pour déduire toutes les combinaisons possibles de ce cas, on procède de la manière suivante:

(a1;(R1|| B || C)) [] (b1; (A || R2 || C)) [] (c1;(A || B || R3))

Avec: R1= a2; a3

R2= b2; b3

R3= c2; c3

- (A [] B) || C = (A || C) [] (B || C)

  Et la symétrique : C || (A [] B) = (C || A) [] (C || B)

Supposons qu'on a une expression d'interface qui contient le parallélisme suivant (tiré du processus `auctionService`):

```
parallel( _in(:seller, :provide, :sellerData),
          _in(:buyer, :provide, :buyerData) )
```

En assertion de traces, on aura donc:

```
if
::canal_seller_In ? provide,  sellerData ;
  canal_buyer_In ? provide,  buyerData ;
::canal_buyer_In ? provide,  buyerData ;
  canal_seller_In ? provide,  sellerData ;
fi;
```

Comme autre exemple, l'expression d'interface du processus `LoanFlow` (il a été extrait du manuel d'instruction d'Oracle BPEL Process Manager (*Oracle BPEL Process Manager 2.0 Quick Start Tutorial*)) est la suivante :

```
canal_client_In ? initiate,  input;
canal_creditRatingService_Out ! process,  crInput;
canal_creditRatingService_In ? process,  crOuput;
( canal_UnitedLoanService_Out ! initiate,
                                  loanApplication;
  canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                  loanOffer1
||
  canal_StarLoanService_Out ! initiate,  loanApplication;
  canal_StarLoanService_In ? onResult_StarLoan,
                                 loanOffer2
);
canal_client_Out ! onResult,  selectedLoanOffer
```

On remarque que nous avons ici deux séquences d'opérations qui peuvent s'exécuter en parallèle :

La 1ère séquence se compose des deux messages (`initiate`, `loanApplication`) et (`onResult_UnitedLoan`, `loanOffer1`).

La 2ème séquence se compose des deux messages (`initiate`, `loanApplication`) et (`onResult_StarLoan`, `loanOffer2`).

En appliquant les règles citées ci-haut, on obtient une assertion de traces qui décrit tous les cas possibles générés à partir de cette concurrence. Au niveau de la partie concurrente du processus, cette assertion de traces décrit la chose suivante : si le processus commence par envoyer le message (`initiate`, `loanApplication`) sur le canal `UnitedLoanService`, il peut soit recevoir le message (`onResult_UnitedLoan`, `loanOffer1`) et par la suite il n'aura pas de choix sauf d'exécuter la deuxième séquence, soit envoyer le message (`initiate`, `loanApplication`) sur le canal `StarLoanService`, et par la suite, il aura deux choix possibles : recevoir le message (`onResult_UnitedLoan`, `loanOffer1`) suivi du message (`onResult_StarLoan`, `loanOffer2`) ou bien inversement. Le processus peut

également commencer par envoyer le message (`initiate, loanApplication`) sur le canal `StarLoanService`, dans ce cas on aura le même résultat, comme le montre la trace qui suit.

```
trace {
  canal_client_In ? initiate,  input;
  canal_creditRatingService_Out ! process,  crInput;
  canal_creditRatingService_In ? process,  crOuput;
  if
  ::canal_UnitedLoanService_Out ! initiate,  loanApplication;
    if
    ::canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                   loanOffer1;
      canal_StarLoanService_Out ! initiate,  loanApplication;
      canal_StarLoanService_In ? onResult_StarLoan,loanOffer2;
    ::canal_StarLoanService_Out ! initiate,loanApplication;
      if
      ::canal_StarLoanService_In ? onResult_StarLoan,
                                   loanOffer2;
        canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                   loanOffer1;
      ::canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                   loanOffer1;
        canal_StarLoanService_In ?  onResult_StarLoan,
                                   loanOffer2;
      fi;
    fi;
  ::canal_StarLoanService_Out ! initiate,  loanApplication;
    if
    ::canal_StarLoanService_In ? onResult_StarLoan,loanOffer2;
      canal_UnitedLoanService_Out ! initiate, loanApplication;
      canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                   loanOffer1;
    ::canal_UnitedLoanService_Out ! initiate, loanApplication;
      if
      ::canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                   loanOffer1;
        canal_StarLoanService_In ? onResult_StarLoan,
                                   loanOffer2;
      ::canal_StarLoanService_In ? onResult_StarLoan,
                                   loanOffer2;
        canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                   loanOffer1;
      fi;
    fi;
  fi;
  canal_client_Out ! onResult,  selectedLoanOffer;
}
```

Reste à signaler le cas suivant : supposons qu'une expression d'interface est écrite comme suit : a;b [] a;c. Nous remarquons qu'elle est non-déterministe. Or, notre outil n'élimine pas ce type de non-déterminisme. Une telle expression ne pourra donc pas être traitée par notre outil. Par contre, signalons qu'ici cette expression peut être exprimée de façon équivalente comme suit : a; (b [] c).

5.2.2.5 Répétition

Pour faciliter l'écriture des expressions d'interface spécifiant une répétition, nous avons utilisé des constructeurs auxiliaires que nous avons catégorisés selon le type de répétition voulu, par exemple, `zeroOrMore`, `oneOrMore`, etc. Pour convertir un bloc répétitif en assertion de traces, nous avons utilisé la boucle `do`. Les objets contenus dans la boucle seront également convertis en assertion de traces, cela en faisant appel à la fonction `to_trace` de chacun d'eux.

5.3 Conclusion

Dans ce chapitre, nous avons présenté nos deux traducteurs qui permettent d'obtenir à partir d'une spécification d'un processus *BPEL* un modèle *Promela*, et à partir d'une spécification d'expression d'interface d'obtenir l'assertion de traces correspondante. La méthode que nous avons utilisée pour l'élimination de la concurrence prend un temps exponentiel, mais, cela n'affecte pas notre travail vu son utilisation sur des descriptions de petites tailles.

Dans le chapitre qui suit, nous allons présenter quelques exemples que nous avons vérifiés avec notre logiciel.

CHAPITRE VI : ÉTUDE DE CAS

Dans ce chapitre, nous présentons quelques exemples de processus d'affaire *BPEL* que nous avons testés avec notre logiciel. Nous allons détailler deux exemples et le reste sera présenté brièvement sous forme d'un tableau. Pour chacun des deux exemples, nous allons décrire la tâche effectuée par le processus en question, puis présenter son environnement. Par la suite, nous allons présenter l'expression d'interface du processus suivie de l'assertion de traces correspondante. Finalement, nous allons présenter le résultat de la vérification avec l'outil *Spin*.

6.1 Le processus LoanFlow

Le processus `LoanFlow` permet à partir d'une commande du client d'invoquer deux partenaires qui répondent chacun par une proposition de prêt. Le processus choisit ensuite le prêt le plus bas et le renvoie comme réponse au client.

6.1.1 Description du processus LoanFlow

Comme le montre la figure 6.1, qui est une représentation simplifiée du processus `LoanFlow` dont le code *BPEL* est présenté en Annexe D, l'exécution du processus `LoanFlow` est amorcée par la réception du message (`initiate, input`). Une fois qu'il reçoit ce message, il invoque son partenaire `creditRatingService` (pour obtenir l'estimation), qui à son tour répond par le message (`process, crOutput`). Par la suite, le processus `LoanFlow` invoque d'une manière asynchrone et concurrente ses deux partenaires `UnitedLoanService` et `StarLoanService` en envoyant à chacun le message (`initiate, loanApplication`), lesquels répondent par les messages (`onResult, loanOffer1`) et (`onResult, loanOffer2`) respectivement. Finalement, le processus sélectionne l'offre la plus basse des deux qu'il a reçu et répond au `client` par le message (`onResult, selectedLoanOffer`).

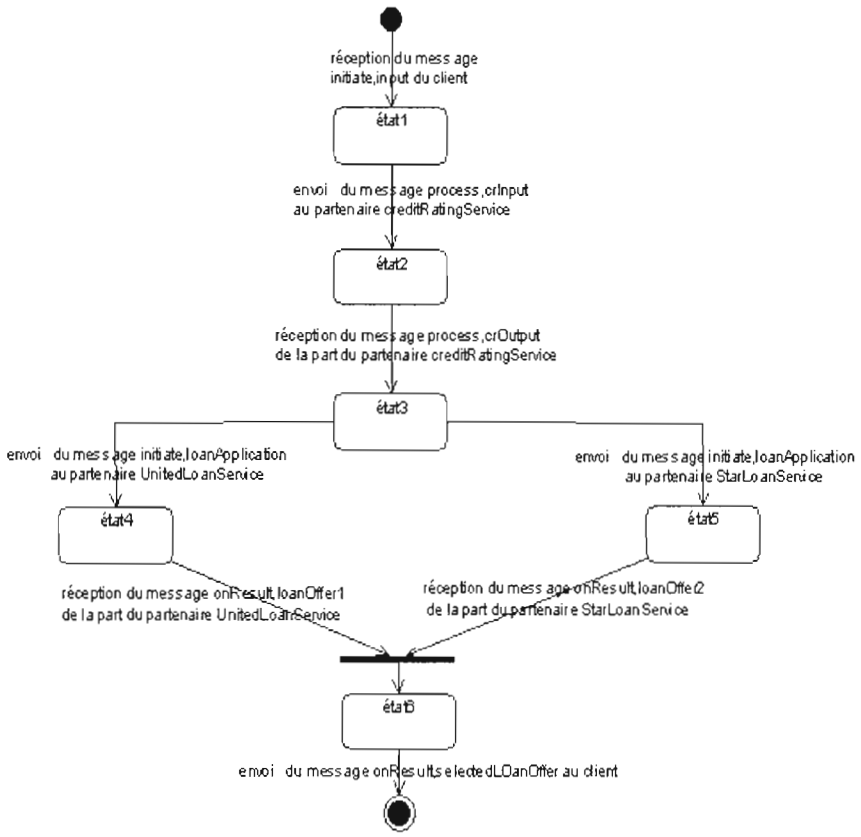


Figure 6.1 - Représentation graphique du processus `LoanFlow`

6.1.2 Fermeture du processus LoanFlow

Le processus `LoanFlow` interagit avec quatre partenaires `client`, `creditRatingService`, `UnitedLoanService` et `StarLoanService`, comme le montre la figure 6.2 qui représente le diagramme de contexte de ce processus.

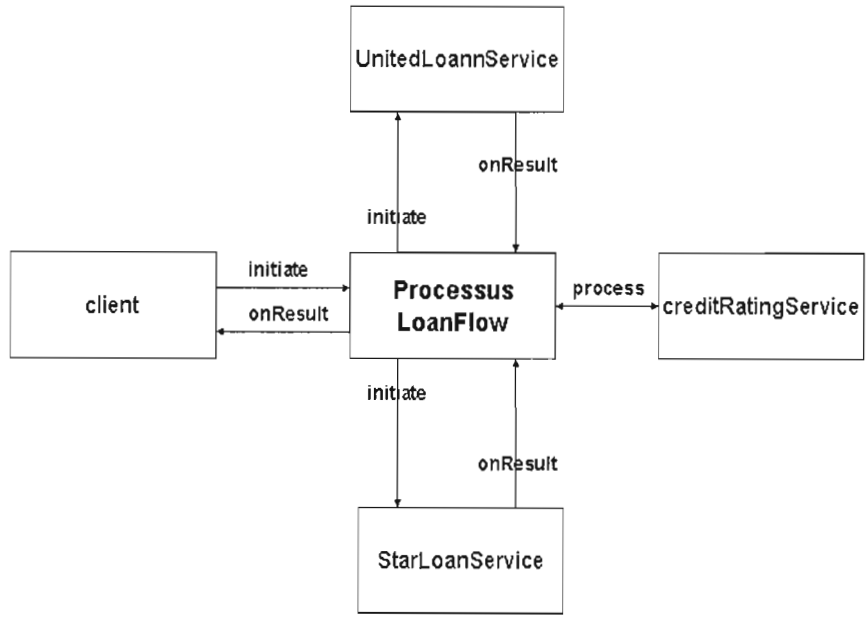


Figure 6.2 - Diagramme de contexte du processus `LoanFlow`

Nous devons spécifier en *Promela* cet environnement, dont le comportement est décrit par la figure 6.3. Chacun de ces partenaires est représenté comme étant un processus *Promela*. Le corps de chacun d'eux dépend de ce que le processus `LoanFlow` attend de lui. Nous avons pris en considération tous les cas possibles en utilisant les affectations non déterministes des variables. Ces processus *Promela* pourraient aussi être obtenus avec notre outil dans le cas où nous aurions leurs spécifications *BPEL*. Cet environnement ainsi que le modèle *Promela* du processus `LoanFlow` sont présentés à l'annexe F.

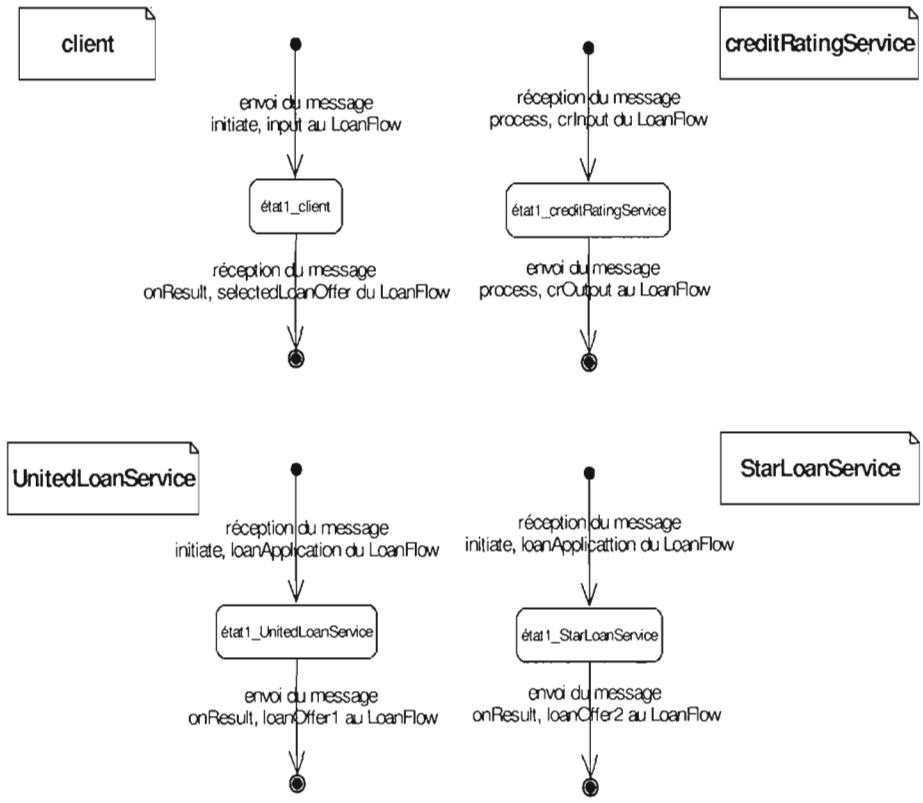


Figure 6.3 - Représentation graphique de l'environnement du processus `LoanFlow`

6.1.3 Expression d'interface et l'assertion de traces correspondante

L'expression d'interface du processus `LoanFlow` est présentée à la figure 6.4.

```
loanFlow =
   canal_client_In ? initiate,  input;
   canal_creditRatingService_Out ! process,  crInput;
   canal_creditRatingService_In ? process,  crOuput;
   ( canal_UnitedLoanService_Out ! initiate,
                                    loanApplication;
     canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                    loanOfferl
   ||
     canal_StarLoanService_Out ! initiate,  loanApplication;
     canal_StarLoanService_In ? onResult_StarLoan,
                                   loanOffer2
   );
   canal_client_Out ! onResult,  selectedLoanOffer
```

Figure 6.4 - Expression d'interface du processus `LoanFlow`

Elle représente exactement ce qui a été présenté dans la section 6.1 de ce chapitre. Notre outil traduit cette expression en assertion de traces, en éliminant la concurrence comme cela a été expliqué dans la section 5.2.2.4 du chapitre 5 et comme cela est montré à la figure 6.5.

```
trace {
    canal_client_In ? initiate,  input ;
    canal_creditRatingService_Out ! process,  crInput ;
    canal_creditRatingService_In ? process,  crOutput ;
    if
      ::canal_UnitedLoanService_Out ! initiate, loanApplication ;
        if
          ::canal_UnitedLoanService_In ? onResult, loanOfferl ;
            canal_StarLoanService_Out ! initiate,loanApplication;
              canal_StarLoanService_In ? onResult,  loanOffer2 ;
          ::canal_StarLoanService_Out ! initiate,loanApplication;
            if
              ::canal_StarLoanService_In ? onResult, loanOffer2 ;
                canal_UnitedLoanService_In ? onResult,loanOfferl;
              ::canal_UnitedLoanService_In ? onResult,loanOfferl;
                canal_StarLoanService_In ? onResult, loanOffer2 ;
            fi;
        fi;
      ::canal_StarLoanService_Out ! initiate,  loanApplication ;
        if
          ::canal_StarLoanService_In ? onResult,  loanOffer2 ;
            canal_UnitedLoanService_Out!
                                    initiate,loanApplication ;
            canal_UnitedLoanService_In ? onResult, loanOfferl ;
          ::canal_UnitedLoanService_Out !
                                    initiate,loanApplication ;
            if
              ::canal_UnitedLoanService_In ? onResult,loanOfferl;
                canal_StarLoanService_In ? onResult, loanOffer2 ;
              ::canal_StarLoanService_In ? onResult, loanOffer2 ;
                canal_UnitedLoanService_In ?
                                      onResult, loanOfferl;
            fi;
          fi;
      fi;
     canal_client_Out ! onResult,  selectedLoanOffer ;
    }
```

Figure 6.5 - Assertion de traces du processus `LoanFlow`

6.1.4 Le résultat de la vérification avec Spin

La figure 6.6 représente le résultat de la vérification du processus *BPEL* `LoanFlow`. Le signe plus (+) montre les options que nous avons utilisées pour effectuer cette vérification. Dans notre cas, nous avons vérifié si le modèle *Promela* respecte ce qui a été spécifié par l'assertion de traces. Nous avons également vérifié s'il existe des états qui ne peuvent pas

être atteints (`unreachable state`). Le résultat produit par *Spin* nous indique que la taille[1] d'un seul état est de 752 octets, la longueur du chemin d'exécution est de 43, et le nombre total des états est de 1179. Le nombre d'erreurs est 0, ce qui veut dire que l'assertion est satisfaite, et tous les états de fin sont valides, ce qui nous conduit à conclure que le processus respecte sa spécification.

```
Full statespace search for:

 trace assertion         +
 never claim             - (not selected)
 assertion violations    - (disabled by -A flag)
 cycle checks            - (disabled by -DSAFETY)
 invalid end states      +

State-vector 752 byte, depth reached 43, errors: 0
    1179 states, stored
     961 states, matched
    2140 transitions (= stored+matched)
       0 atomic steps
unreached in proctype process_1
 (0 of 4 states)
unreached in proctype process_2
 (0 of 4 states)
unreached in proctype LoanFlow
 (0 of 17 states)
unreached in proctype client
 (0 of 3 states)
unreached in proctype UnitedLoanService
 (0 of 9 states)
unreached in proctype StarLoanService
 (0 of 9 states)
unreached in proctype creditRatingService
 (0 of 3 states)
unreached in proctype :trace:
 (0 of 33 states)
unreached in proctype :init:
 (0 of 6 states)
```

Figure 6.6 - Résultat de la vérification du processus `LoanFlow`

1 La taille d'un état est la quantité de mémoire nécessaire pour stocker l'ensemble des informations qui décrivent un état.

6.1.5 Modification de l'expression d'interface du processus LoanFlow

Supposons qu'on modifie l'expression d'interface du processus `LoanFlow` comme le montre la figure 6.7. Cette nouvelle expression d'interface indique que le processus `LoanFlow` reçoit le message (`initiate`, `input`) deux fois : au début de son initialisation et à la fin avant qu'il envoie une réponse au `client`.

```
loanFlow =
   canal_client_In ? initiate,  input;
   canal_creditRatingService_Out ! process,  crInput;
   canal_creditRatingService_In ? process,  crOuput;
   ( canal_UnitedLoanService_Out ! initiate,
                                    loanApplication;
     canal_UnitedLoanService_In ? onResult_UnitedLoan,
                                    loanOffer1
   ||
     canal_StarLoanService_Out ! initiate,  loanApplication;
     canal_StarLoanService_In ? onResult_StarLoan,
                                   loanOffer2
   );
   canal_client_In ? initiate,  input;
   canal_client_Out ! onResult,  selectedLoanOffer
```

Figure 6.7 - Expression d'interface du processus `LoanFlow` modifiée

Le résultat de la vérification présenté par la figure 6.8 montre que le processus `LoanFlow` ne respecte pas cette expression d'interface modifiée vu que *Spin* signale une erreur. Le résultat de cette vérification montre que certains cas n'ont pas été explorés et que le nombre d'états stockés pour trouver l'erreur est de 39, la taille d'un état est de 744 et la longueur du chemin d'exécution est de 38. Pour conclure, le processus `LoanFlow` ne satisfait pas sa nouvelle spécification.

```
Full statespace search for:
 trace assertion         +
 never claim             - (not selected)
 assertion violations    - (disabled by -A flag)
 cycle checks            - (disabled by -DSAFETY)
 invalid end states      +

 State-vector 744 byte, depth reached 38, errors: 1
       39 states, stored
        0 states, matched
       39 transitions (= stored+matched)
        0 atomic steps
 hash conflicts: 0 (resolved)

 Stats on memory usage (in Megabytes):
 0.029       equivalent memory usage for states
 (stored*(State-vector + overhead))
 0.299       actual memory usage for states (unsuccessful
 compression: 1024.72%)
       State-vector as stored = 7661 byte + 4 byte overhead
 2.097       memory used for hash table (-w19)
 0.280       memory used for DFS stack (-m10000)
 0.091       other (proc and chan stacks)
 0.095       memory lost to fragmentation
 2.582       total actual memory usage

 unreached in proctype process_1
       (0 of 4 states)
 unreached in proctype process_2
       (0 of 4 states)
 unreached in proctype LoanFlow
       line 96, "pan_in", state 11, "selectedLoanOffer.payload
       = loanOffer2.payload"
       line 101, "pan_in", state 17, "-end-"
       (2 of 17 states)
 unreached in proctype client
       line 110, "pan_in", state 3, "-end-"
       (1 of 3 states)
 unreached in proctype UnitedLoanService
       line 120, "pan_in", state 5, "loanOffer1.payload = 0"
       line 118, "pan_in", state 6, "(1)"
       line 118, "pan_in", state 6, "(1)"
       (2 of 9 states)
 unreached in proctype StarLoanService
       line 133, "pan_in", state 5, "loanOffer2.payload = 0"
       line 131, "pan_in", state 6, "(1)"
       line 131, "pan_in", state 6, "(1)"
       (2 of 9 states)
 unreached in proctype creditRatingService
       (0 of 3 states)
```

```
unreached in proctype :trace:
      line 160, "pan_in", state 10,
"canal_UnitedLoanService_In?onResult,loanOffer1.payload"
      line 162, "pan_in", state 12,
"canal_StarLoanService_In?onResult,loanOffer2.payload"
      line 158, "pan_in", state 13,
"canal_StarLoanService_In?onResult,loanOffer2.payload"
      line 158, "pan_in", state 13,
"canal_UnitedLoanService_In?onResult,loanOffer1.payload"
      line 168, "pan_in", state 19,
"canal_UnitedLoanService_Out!initiate,loanApplication.payload
.ch"
      line 169, "pan_in", state 20,
"canal_UnitedLoanService_In?onResult,loanOffer1.payload"
      line 173, "pan_in", state 23,
"canal_StarLoanService_In?onResult,loanOffer2.payload"
      line 175, "pan_in", state 25,
"canal_UnitedLoanService_In?onResult,loanOffer1.payload"
      line 171, "pan_in", state 26,
"canal_UnitedLoanService_In?onResult,loanOffer1.payload"
      line 171, "pan_in", state 26,
"canal_StarLoanService_In?onResult,loanOffer2.payload"
      line 166, "pan_in", state 28,
"canal_StarLoanService_In?onResult,loanOffer2.payload"
      line 166, "pan_in", state 28,
"canal_UnitedLoanService_Out!initiate,loanApplication.payload
.ch"
      line 180, "pan_in", state 33,
"canal_client_Out!onResult,selectedLoanOffer.payload"
      line 181, "pan_in", state 34, "-end-"
      (11 of 34 states)
unreached in proctype :init:
      line 189, "pan_in", state 6, "-end-"
      (1 of 6 states)
```

Figure 6.8 - Résultat de la vérification du processus `LoanFlow` pour l'expression d'interface modifiée

6. 2 Le processus decValMachine

Le processus `decValMachine` décrit les opérations d'incrémentation et décrémentation effectuées par une machine selon les messages qu'elle reçoit. Elle renvoie le résultat des opérations au client lorsqu'il lui signale qu'il a terminé.

6.2.1 Description du processus decValMachine

Comme le montre le figure 6.9, qui est une représentation simplifiée du processus `decValMachine` dont le code *BPEL* est présenté en annexe E, l'exécution du processus est amorcée par la réception du message (`initiate, x`) envoyé par le partenaire `client`. Par la suite, le processus `decValMachine` rentre dans une boucle qui sera achevée par la réception du message (`val, dummy`) de son unique partenaire `client`, sinon il reçoit de lui, soit le message (`inc, x`) qui va lui permettre d'effectuer une opération d'incrémentation ou bien le message (`dec, x`) pour effectuer une opération de décrémentation. Finalement, il répond à son partenaire par le message (`end, r`) sachant que `r` contient le résultat des opérations qu'il a effectuées.

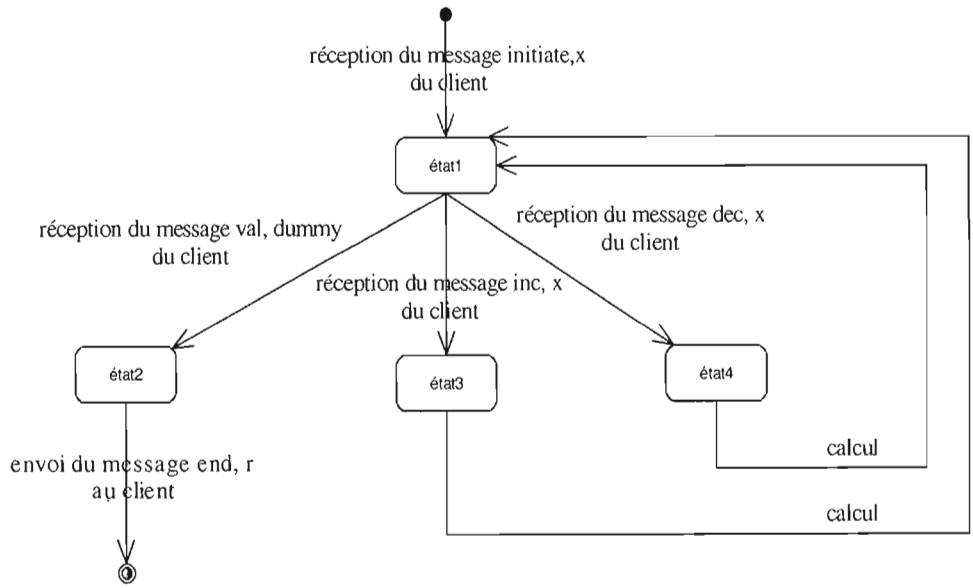


Figure 6.9 - Représentation graphique du processus `decValMachine`

6.2.2 Fermeture du processus decValMachine

L'environnement du processus `decValMachine` est composé d'un seul processus `client` comme le montre la figure 6.10.

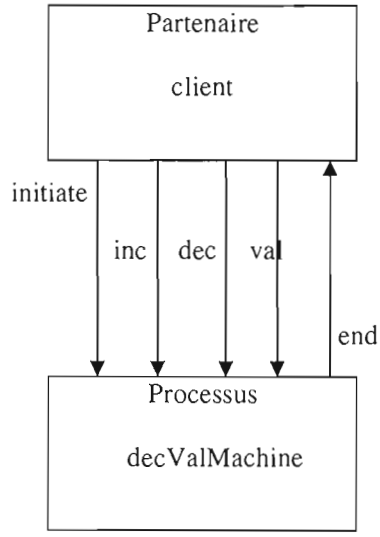


Figure 6.10 - Diagramme de contexte du processus `decValMachine`

Comme nous l'avons déjà indiqué au chapitre 5, nous pouvons obtenir le modèle *Promela* correspondant du partenaire de ce processus à partir de sa spécification *BPEL*. La figure 6.11 est une représentation abstraite de cet environnement. L'annexe G représente le modèle *Promela* du processus `decValMachine` et de son partenaire.

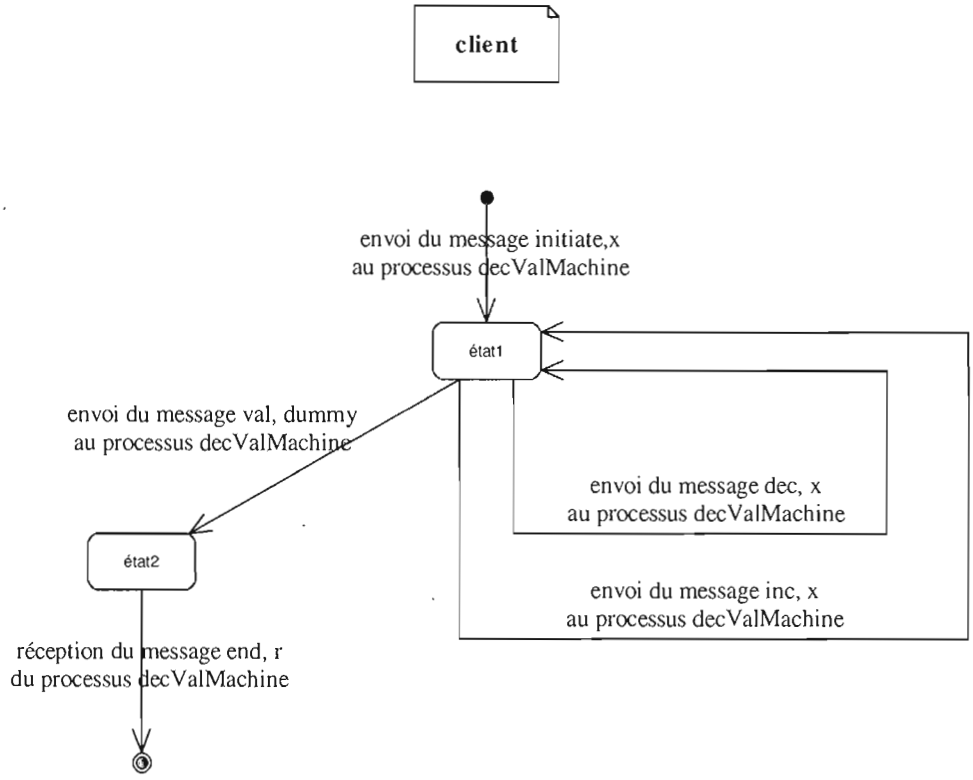


Figure 6.11 - Représentation graphique du comportement du client du processus `decValMachine`

6.2.3 Expression d'interface et l'assertion de traces correspondante

L'expression d'interface du processus `decValMachine` est présentée à la figure 6.12.

```
decValMachine =
  canal_client_In ? initiate,  x, _ ;
  (  canal_client_In ? inc,  x, _
  [] canal_client_In ? dec,  x, _
  )+;
 canal_client_In ? val,  _ , dummy
 canal_client_Out ! end,  r
```

Figure 6.12 - Expression d'interface du processus `decValMachine`

Cette expression d'interface montre que le processus `decValMachine` reçoit de son unique partenaire en premier lieu le message (`initiate, x`). Ensuite, il reçoit soit le message (`inc,x`) ou le message (`dec,x`) qui vont lui permettre de rester dans la boucle sinon il reçoit (`val,dummy`) et sort de la boucle. Puis, il répond à son partenaire par le message (`end,r`). Notre outil traduit cette expression en assertion de traces, comme le montre la figure 6.13.

```
trace {
  canal_client_In ? initiate, x , _ ;
  do
  ::  canal_client_In ? inc, x , _ ;
  ::  canal_client_In ? dec, x , _ ;
  ::  canal_client_In ? val, _ , dummy ;  break ;
  od;
  canal_client_Out ! end, r  ;
}
```

Figure 6.13 - Assertion de traces du processus `decValMachine`

6.2.4 Le résultat de la vérification avec Spin

La figure 6.14 présente le résultat de la vérification du processus `decValMachine`. Le processus `decValMachine` respecte son expression d'interface puisqu'il satisfait l'assertion de traces vue que le nombre d'erreurs retourné par le vérificateur *Spin* est 0. Tous les états ont été explorés. Le résultat généré pas *Spin* nous montre que le nombre d'états stockés pour accomplir cette vérification est de 23036, la taille d'un état est de 260 octets et la longueur du chemin d'exécution est de 7672. Pour conclure, le processus `decValMchaine` respecte sa spécification.

```
Full statespace search for:
 trace assertion         +
 never claim             - (not selected)
 assertion violations    - (disabled by -A flag)
 cycle checks            - (disabled by -DSAFETY)
 invalid end states      +

 State-vector 260 byte, depth reached 7672, errors: 0
    23036 states, stored
    23015 states, matched
    46051 transitions (= stored+matched)
        0 atomic steps
 hash conflicts: 348 (resolved)

 Stats on memory usage (in Megabytes):
 6.082       equivalent memory usage for states
 (stored*(State-vector + overhead))
 6.142       actual memory usage for states (unsuccessful
 compression: 101.00%)
 State-vector as stored = 263 byte + 4 byte overhead
 2.097       memory used for hash table (-w19)
 0.280       memory used for DFS stack (-m10000)
 0.078       other (proc and chan stacks)
 0.101       memory lost to fragmentation
 8.419       total actual memory usage

 unreached in proctype decValMachine
       (0 of 25 states)
 unreached in proctype client
       (0 of 10 states)
 unreached in proctype :trace:
       (0 of 10 states)
 unreached in proctype :init:
       (0 of 3 states)
```

Figure 6.14 - Résultat de la vérification du processus `decValMachine`

6.2.5 Modification de l'expression d'interface du processus decValMachine

Supposons que nous modifions l'expression d'interface du processus `decValMachine` comme le montre la figure 6.15. Cette modification indique que lorsque le processus `decValMachine` rentre dans la boucle, il ne peut recevoir que le message (`inc`, `x`) qui lui permet de rester dans la boucle, ou bien le message (`val`, `dummy`) qui va lui permettre de sortir pour envoyer par la suite une réponse à son partenaire.

```
      canal_client_In ? initiate,  x, _ ;
      (  canal_client_In ? inc,  x, _
      [] canal_client_In ? val,  _ , dummy
      )+;
    canal_client_Out ! end,  r
```

Figure 6.15 - Expression d'interface du processus `decValMachine` modifiée

Le résultat de la vérification présenté par la figure 6.16 montre que le processus `decValMachine` ne respecte pas l'expression d'interface de la figure 6.13 vu que *Spin* signale une erreur. Ce résultat indique que certains cas n'ont pas été explorés, et que le nombre d'états stockés pour trouver l'erreur est de 59, la taille d'un état de 260 et la longueur du chemin d'exécution est de 57.

```
Full statespace search for:
  trace assertion        +
  never claim            - (not selected)
  assertion violations   - (disabled by -A flag)
  cycle checks           - (disabled by -DSAFETY)
  invalid end states     +

State-vector 260 byte, depth reached 57, errors: 1
      59 states, stored
       1 states, matched
      60 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.016    equivalent memory usage for states (stored*(State-
vector + overhead))
0.303    actual memory usage for states (unsuccessful
compression: 1942.96%)
  State-vector as stored = 5125 byte + 4 byte overhead
2.097    memory used for hash table (-w19)
0.280    memory used for DFS stack (-m10000)
0.084    other (proc and chan stacks)
0.098    memory lost to fragmentation
2.582    total actual memory usage
unreached in proctype decValMachine
  line 50, "pan_in", state 9, "s.idpart = (s.idpart-
  x.idpart)"
  line 55, "pan_in", state 15, "s.idpart = 1"
  line 56, "pan_in", state 16, "r.idpart = s.idpart"
  line 44, "pan_in", state 17, "canal_client_In?
  inc,x.idpart,_"
  line 44, "pan_in", state 17, "canal_client_In?
  dec,x.idpart,_"
  line 44, "pan_in", state 17, "canal_client_In?
  val,_,dummy.message.ch"
  line 60, "pan_in", state 24, "canal_client_Out!
  end,r.idpart"
  line 62, "pan_in", state 25, "-end-"
  (6 of 25 states)
unreached in proctype client
  line 72, "pan_in", state 9, "canal_client_Out?end,r.idpart"
  line 73, "pan_in", state 10, "-end-"
  (2 of 10 states)
unreached in proctype :trace:
  line 81, "pan_in", state 8, "canal_client_Out!end,r.idpart"
  line 82, "pan_in", state 9, "-end-"
  (2 of 9 states)
unreached in proctype :init: (0 of 3 states)
```

Figure 6.16 - Résultat de la vérification du processus `decValMachine` pour l'expression d'interface modifiée

6.3 Autres processus

Le tableau ci-dessous présente quelques processus que nous avons traités. Pour chaque processus, on trouve son expression d'interface et son assertion de traces correspondante. Les résultats de vérification avec l'outil *Spin* ont montré que ces processus respectent leurs interfaces.

Expression d'interface	Assertion trace correspondante

Processus simple:
 	
canal_caller_In ?
gimmeQuote, request;
canal_provider_Out !
getQuote,invocationreques;
canal_provider_In ?
getQuote,invocationresponse;
canal_caller_Out !
gimmeQuote, response
	trace{
	canal_caller_In ?
	gimmeQuote, request ;
	canal_provider_Out !
	getQuote, invocationrequest;
	canal_provider_In ?
	getQuote,invocationresponse;
	canal_caller_Out !
	gimmeQuote, response ;
	}

Processus flightBookingFlow:
 	
canal_client_In ?
initiate, input, _, _;
canal_client_Out !
onOffer, offer,
varFactice5; (canal_client_In ?
approve, varFactice2,
approved,varFactice4
[]
canal_client_In ? cancel,
cancel,varFactice2,
varFactice3, canceled
);
canal_client_Out ! onResult,
varFacticel, output
	trace{
	canal_client_In ?
	initiate, input, _, _ ;
	canal_client_Out !
	onOffer, offer,varFactice5;
	if
	::canal_client_In ?
	approve, varFactice2,
	approved, varFactice4 ;
	::canal_client_In ? cancel,
	cancel, varFactice2,
	varFactice3, canceled ;
	fi;
	canal_client_Out !
	onResult,varFacticel,output;
	}

Processus CreditFlow:
 	
canal_client_In ?
initiate, input;
canal_creditRatingService_Out !
process, input;
(canal_creditRatingService_In ?
process, output, _
[]
canal_creditRatingService_In ?
NegativeCredit, _, fault
);
canal_client_Out !
onResult, output
	trace{
	canal_client_In ?
	initiate, input ;
	canal_creditRatingService_Out !
	process, input ;
	if
	::canal_creditRatingService_In?
	process, output, _ ;
	::canal_creditRatingService_In?
	NegativeCredit, _, fault ;
	fi;
	canal_client_Out !
	onResult, output ;
	}

Processus ordersProcess:

```
canal_ordering_In ?
       order, order1;
(canal_warehouse_Out !
       order, order1;
  canal_warehouse_In ?
       order, invoice
  []
  canal_warehouse_Out !
       schedule, order1;
 (canal_warehouse_In ?
       receive_not, invoice
  []
  canal_Warehouse_Out !
       order, order1;
  canal_Warehouse_In ?
       order, invoice
  )
);
canal_ordering_Out !
      order, invoice
```

```
trace{
  canal_ordering_In ?
         order, order1;
  if
  ::canal_warehouse_Out !
          order, order1 ;
    canal_warehouse_In ?
          order, invoice ;
  ::canal_warehouse_Out !
           schedule,  order1 ;
      if
      ::canal_warehouse_In ?
           receive_not, invoice ;
      ::canal_Warehouse_Out !
           order,  order1 ;
        canal_Warehouse_In ?
           order,  invoice ;
      fi;
  fi;
  canal_ordering_Out !
         order, invoice ;
 }
```

CONCLUSION

Le travail effectué dans le cadre de ce mémoire de maîtrise a donné naissance à notre logiciel qui permet de valider un processus exécutable *BPEL* par rapport à une spécification de son interface. La spécification des processus d'affaire électroniques est décrite en langage *BPEL*, qui vise à spécifier l'orchestration du processus, c'est-à-dire, comment le processus coopère avec ses partenaires pour accomplir un service donné. La vérification du processus *BPEL* est effectuée par rapport à son expression d'interface représentant les messages qu'il échange avec ses partenaires. La technique que nous avons utilisée pour la vérification est le la vérification de modèles (*model checking*) qui permet de détecter des erreurs pour les corriger à une étape précoce du cycle de vie du processus.

La procédure que nous avons suivie pour effectuer cette vérification consiste à traduire le processus *BPEL* en modèle *Promela* correspondant, puis à le fermer, c'est-à-dire, inclure dans le modèle *Promela* l'environnement (les autres processus) avec lequel ce processus interagit. Ensuite, on génère à partir de la spécification d'interface l'assertion de traces correspondante. Enfin, on lance la vérification avec *Spin* et on analyse le résultat. Signalons que les travaux décrits dans ce mémoire ont donné lieu à une publication [39].

Différentes équipes de chercheurs ont considéré le problème de la vérification de processus *BPEL*. Une multitude de méthodes d'abstraction a été proposée et une variété d'outils de *model checking* a été utilisée. Notre travail diffère de ces travaux au niveau des parties suivantes :

- Pour traduire un processus *BPEL* en modèle *Promela*, nous l'avons fait directement, sans passer par une représentation intermédiaire, et ce à l'aide de l'API BPWS4J.

- Les équipes qui ont utilisé l'outil *Spin* comme vérificateur ont exprimé les propriétés à vérifier en logique LTL. Ces équipes n'ont pas mentionné comment ils ont fait pour aboutir à ces propriétés LTL. De plus, ces propriétés semblent souvent porter sur l'état interne du processus. Dans notre cas, on passe à notre logiciel le processus *BPEL* ainsi que son expression d'interface avec l'objectif de vérifier s'il la respecte. Cette expression d'interface représente l'ordre et les types des messages que le processus échange avec son environnement, c'est-à-dire, ses comportements possibles tels que perçus par un observateur externe.

- Les expressions d'interface sont traduites en assertion de traces, qui permettent la surveillance d'évènements d'envoi et de réception d'une manière relativement simple et qui permettent de décrire des expressions de chemins d'une manière plus abstraite.

Suite au travail que nous avons effectué au niveau de ce mémoire, il reste encore des pistes à explorer :

- Ajouter des éléments *BPEL* que nous n'avons pas traités comme par exemple le traitement des erreurs globales.

- Essayer de traduire les expressions d'interface en propriétés LTL ou en *never claim*.

- Travailler sur d'autres types de propriétés à vérifier.

- Lorsque la propriété n'est pas vérifiée, *Spin* retourne un contre-exemple. Alors, essayer à partir de ce contre-exemple de trouver la partie correspondante en *BPEL* qui cause l'erreur, puis proposer une correction.

ANNEXE A : EXEMPLE DE FICHIER WSDL

Cette annexe représente un exemple de définition d'un fichier WSDL d'un simple service fournissant les estimations d'un stock [6].

```
<?xml version="1.0"?>
<definitions name="StockQuote"
          targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
        <schema
            targetNamespace="http://example.com/stockquote.xsd"
                    xmlns="http://www.w3.org/2000/10/XMLSchema">
          <element name="TradePriceRequest">
             <complexType>
                 <all>
                     <element name="tickerSymbol1"
                              type="string"/>
                 </all>
             </complexType>
          </element>
          <element name="TradePrice">
             <complexType>
                 <all>
                     <element name="price" type="float"/>
                 </all>
             </complexType>
          </element>
       </schema>
    </types>
    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>
    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
    <binding name="StockQuoteSoapBinding"
             type="tns:StockQuotePortType">
        <soap:binding style="document"
              transport="http://schemas.xmlsoap.org/soap/http"/>
         <operation name="GetLastTradePrice">
```

```
            <soap:operation
             soapAction="http://example.com/GetLastTradePrice"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort"
              binding="tns:StockQuoteBinding">
            <soap:address
                  location="http://example.com/stockquote"/>
        </port>
    </service>
</definitions>
```

ANNEXE B : SPÉCIFICATION BPEL DU PROCESSUS AUCTIONSERVICE

Cette annexe représente le code *BPEL* du processus `auctionService` qui permet d'effectuer une opération d'enchères entre un vendeur et un client en invoquant son partenaire `auctionRegistrationService` qui enregistre la vente [1].

```
<process name="auctionService"
         targetNamespace="http://www.auction.com"
         containerAccessSerializable="no"
         xmlns:as="http://www.auction.com/wsdl/auctionService"
         xmlns:sref="http://schemas.xmlsoap.org/ws/2002/07/
         service-reference/"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
         process/">

  <partnerLinks>
    <partnerLink name="seller"
             serviceLinkType="as:sellerAuctionHouseLT"
             myRole="auctionHouse" partnerRole="seller"/>
    <partnerLink name="buyer"
             serviceLinkType="as:buyerAuctionHouseLT"
             myRole="auctionHouse" partnerRole="buyer"/>
    <partnerLink name="auctionRegistrationService"
             serviceLinkType=
             "as:auctionHouseAuctionRegistrationServiceLT"
             myRole="auctionHouse"
             partnerRole="auctionRegistrationService"/>
  </partnerLinks>

  <variables>
    <variable name="sellerData" messageType="as:sellerData"/>
    <variable name="sellerAnswerData"
              messageType="as:sellerAnswerData"/>
    <variable name="buyerData" messageType="as:buyerData"/>
    <variable name="buyerAnswerData"
              messageType="as:buyerAnswerData"/>
    <variable name="auctionRegistrationData"
              messageType="as:auctionDetails"/>
    <variable name="auctionRegistrationResponse"
               messageType="as:auctionRegAnswer"/>
  </variables>

  <correlationSets>
    <corellationSet name="auctionIdentification"
                    properties="as:auctionId"/>
  </correlationSets>

  <sequence>
    <flow>
      <receive name="acceptSellerInformation"
```

```
            partnerLink="seller"
           portType="as:sellerPT"
           operation="provide"
           variable="sellerData"
           createInstance="yes">
    <correlations>
      <correlation set="tns:auctionIdentification"
                   initiation="yes"/>
    </correlations>
  </receive>
  <receive name="acceptBuyerInformation"
           partnerLink="buyer"
           portType="as:buyerPT"
           operation="provide"
           variable="buyerData"
           createInstance="yes">
    <correlations>
      <correlation set="tns:auctionIdentification"
                   initiation="yes"/>
    </correlations>
  </receive>
</flow>
<assign>
  <copy>
    <from partnerLink="seller"/>
    <to partnerLink="auctionRegistrationService"/>
  </copy>
</assign>
<assign>
  <copy>
    <from partnerLink="auctionRegistrationService"/>
    <to variable="auctionRegistrationData"
        part="auctionHouseServiceRef"/>
  </copy>
</assign>
 <invoke name="registerAuctionResults"
        partnerLink="auctionRegistrationService"
        portType="as:auctionRegistrationPT"
        operation="process"
        inputVariable="auctionRegistrationData">
  <correlations>
    <correlation set="auctionIdentification"
                 pattern="out"/>
  </correlations>
</invoke>
<receive name="receiveAuctionRegistrationInformation"
         partnerLink="auctionRegistrationService"
         portType="as:auctionRegistrationAnswerPT"
         operation="answer"
         variable="auctionRegistrationResponse">
  <correlations>
    <correlation set="auctionIdentification"/>
  </correlations>
</receive>
<flow>
   <sequence>
    <assign>
       <copy>
```

```
            <from variable="sellerData"
                  part="serviceReference"/>
            <to partnerLink="seller"/>
          </copy>
        </assign>
        <invoke name="respondToSeller"
                partnerLink="seller"
                portType="as:sellerAnswerPT"
                operation="answer"
                inputVariable="sellerAnswerData"/>
      </sequence>
      <sequence>
        <assign>
          <copy>
            <from variable="buyerData"
                  part="serviceReference"/>
            <to partnerLink="buyer"/>
          </copy>
        </assign>
        <invoke name="respondToBuyer"
                partnerLink="buyer"
                portType="as:buyerAnswerPT"
                operation="answer"
                inputVariable="buyerAnswerData"/>
      </sequence>
    </flow>
  </sequence>
</process>
```

ANNEXE C : SPÉCIFICATION BPEL DU PROCESSUS PURCHASEORDER

Cette annexe représente le code *BPEL* du processus `purchaseOrder`. Pour que ce processus réponde à une commande d'achat faite par un client, il invoque ses partenaires : `shipping` pour organiser la livraison, `invoicing` pour effectuer la facturation et `scheduling` pour établir l'horaire [1].

```
<process name="purchaseOrderProcess"
         targetNamespace="http://acme.com/ws-bp/purchase"
         xmlns:lns="http://manufacturing.org/wsdl/purchase"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
         process/">

  <import namespace="http://manufacturing.org/wsdl/purchase"
          location="purchase.wsdl"/>

  <variables>
    <variable name="PO" messageType="lns:POMessage"/>
         <variable name="Invoice"
              messageType="lns:InvMessage"/>
    <variable name="POFault"
              messageType="lns:orderFaultType"/>
    <variable name="shippingRequest"
              messageType="lns:shippingRequestMessage"/>
        <variable name="shippingInfo"
              messageType="lns:shippingInfoMessage"/>
    <variable name="shippingSchedule"
              messageType="lns:scheduleMessage"/>
    </variables>

  <partnerLinks>
    <partnerLink name="purchasing"
             serviceLinkType="lns:purchaseLT"
             myRole="purchaseService"/>
    <partnerLink name="invoicing"
             serviceLinkType="lns:invoiceLT"
             partnerLinkRole="invoiceService"/>
    <partnerLink name="shipping"
             serviceLinkType="lns:shippingLT"
             partnerLinkRole="shippingService"/>
    <partnerLink name="scheduling"
             serviceLinkType="lns:schedulingLT"
             partnerLinkRole="schedulingService"/>
  </partnerLinks>

  <faultHandlers>
    <catch faultName="lns:cannotCompleteOrder"
           faultVariable="POFault">
```

```
    <reply portType="lns:purchasePT"
           operation="sendPurchaseOrder"
           variable="POFault"
           faultName="lns:cannotCompleteOrder"/>
  </catch>
</faultHandlers>

<sequence>
  <receive partnerLink="purchasing"
           portType="lns:purchaseOrderPT"
           operation="sendPurchaseOrder" variable="PO">
  </receive>
  <flow>
    <links>
      <link name="ship-to-invoice"/>
      <link name="ship-to-scheduling"/>
    </links>
    <sequence>
      <assignment>
        <xpath>
          <from variable="PO" part="customerInfo"/>
          <to variable="shippingRequest"
              part="customerInfo"/>
        </xpath>
      </assignment>
      <invoke partnerLink="shipping"
              portType="lns:shippingPT"
              operation="requestShipping"
              inputVariable="shippingRequest"
              outputVariable="shippingInfo">
        <source linkName="ship-to-invoice"/>
      </invoke>
      <receive partnerLink="shipping"
               portType="lns:shippingCallbackPT"
               operation="sendSchedule"
               variable="shippingSchedule">
        <source linkName="ship-to-scheduling"/>
      </receive>
    </sequence>
    <sequence>
      <invoke partnerLink="invoicing"
              portType="lns:computePricePT"
              operation="initiatePriceCalculation"
              inputVariable="PO">
      </invoke>
      <invoke partnerLink="invoicing"
              portType="lns:computePricePT"
              operation="sendShippingPrice"
              inputVariable="shippingInfo">
        <target linkName="ship-to-invoice"/>
      </invoke>
      <receive partnerLink="invoicing"
               portType="lns:invoiceCallbackPT"
               operation="sendInvoice"
               variable="Invoice"/>
    </sequence>
    <sequence>
       <invoke partnerLink="scheduling"
```

```
                portType="lns:schedulingPT"
                operation="requestProduction"
                inputVariable="PO">
            </invoke>
            <invoke partnerLink="scheduling"
                portType="lns:schedulingPT"
                operation="sendShipping"
                inputVariable="shippingSchedule">
              <target linkName="ship-to-scheduling"/>
            </invoke>
          </sequence>
        </flow>
        <reply partnerLink="purchasing" portType="lns:purchasePT"
               operation="sendPurchaseOrder"
               variable="Invoice"/>
    </sequence>
</process>
```

ANNEXE D : SPÉCIFICATION BPEL DU PROCESSUS LOANFLOW

Cette annexe représente la spécification *BPEL* du processus `LoanFlow` présenté au chapitre 6. Ce processus a été extrait du manuel d'instruction d'Oracle *BPEL* Process Manager (*Oracle BPEL Process Manager 2.0 Quick Start Tutorial*).

```
<process name="LoanFlow"
         targetNamespace="http://samples.otn.com"
         suppressJoinFailure="yes"
         xmlns:tns="http://samples.otn.com"
         xmlns:services="http://services.otn.com"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
         process/">


  <partnerLinks>
    <partnerLink name="client"
             partnerLinkType="tns:LoanFlow"
             partnerRole="LoanFlowRequester"
             myRole="LoanFlowProvider"/>
    <partnerLink name="creditRatingService"
             partnerLinkType="services:CreditRatingService"
             partnerRole="CreditRatingServiceProvider"/>
    <partnerLink name="UnitedLoanService"
             partnerLinkType="services:LoanService"
             myRole="LoanServiceRequester"
             partnerRole="LoanServiceProvider"/>
    <partnerLink name="StarLoanService"
             partnerLinkType="services:LoanService"
             myRole="LoanServiceRequester"
             partnerRole="LoanServiceProvider"/>
  </partnerLinks>

  <variables>
    <variable name="input"
              messageType="tns:LoanFlowRequestMessage"/>
    <variable name="crInput"
      messageType="services:CreditRatingServiceRequestMessage"/>
    <variable name="crOutput"
     messageType="services:CreditRatingServiceResponseMessage"/>
    <variable name="crError"
    messageType="services:CreditRatingServiceFaultMessage"/>
    <variable name="loanApplication"
              messageType="services:LoanServiceRequestMessage"/>
    <variable name="loanOffer1"
               messageType="services:LoanServiceResultMessage"/>
    <variable name="loanOffer2"
               messageType="services:LoanServiceResultMessage"/>
     <variable name="selectedLoanOffer"
```

```
            messageType="tns:LoanFlowResultMessage"/>
</variables>

<sequence>
  <receive name="receiveInput" partnerLink="client"
           portType="tns:LoanFlow"
           operation="initiate" variable="input"
           createInstance="yes"/>
  <scope name="GetCreditRating">
     <faultHandlers>
      <catch faultName="services:NegativeCredit"
          faultVariable="crError">
          <assign>
              <copy>
                  <from expression="number(-1000)"/>
                  <to variable="input" part="payload"
                      query="/loanApplication/creditRating"/>
              </copy>
          </assign>
      </catch>
    </faultHandlers>
    <sequence>
    <assign>
      <copy>
          <from variable="input" part="payload"
               query="/loanApplication/SSN"/>
          <to variable="crInput" part="payload"
              query="/ssn"/>
      </copy>
    </assign>
       <invoke name="invokeCR"
              partnerLink="creditRatingService"
              portType="services:CreditRatingService"
              operation="process"
              inputVariable="crInput"
              outputVariable="crOutput"/>
       <assign>
            <copy>
              <from variable="crOutput" part="payload"
                    query="/rating"/>
              <to variable="input" part="payload"
                  query="/loanApplication/creditRating"/>
          </copy>
       </assign>
    </sequence>
  </scope>
<scope name="GetLoanOffer">
    <sequence>
          <assign>
              <copy>
                  <from variable="input" part="payload"/>
                  <to variable="loanApplication"
                     part="payload"/>
              </copy>
          </assign>
          <flow>
             <sequence>
                    <invoke name="invokeUnitedLoan"
```

```
                    partnerLink="UnitedLoanService"
                    portType="services:LoanService"
                    operation="initiate"
                    inputVariable="loanApplication"/>
                <receive name="receive_invokeUnitedLoan"
                    partnerLink="UnitedLoanService"
                    portType="services:LoanServiceCallback"
                    operation="onResult"
                    variable="loanOffer1"/>
            </sequence>
            <sequence>
                <invoke name="invokeStarLoan"
                        partnerLink="StarLoanService"
                    portType="services:LoanService"
                    operation="initiate"
                    inputVariable="loanApplication"/>
                 <receive name="receive_invokeStarLoan"
                    partnerLink="StarLoanService"
                    portType="services:LoanServiceCallback"
                    operation="onResult"
                    variable="loanOffer2"/>
            </sequence>
          </flow>
    </sequence>
   </scope>
   <scope name="SelectOffer">
    <switch>
        <case
            condition="bpws:getVariableData
            ('loanOffer1','payload','/loanOffer/APR') >
            bpws:getVariableData('loanOffer2','payload',
            '/loanOffer/APR') ">
              <assign>
              <copy>
              <from variable="loanOffer2" part="payload"/>
              <to variable="selectedLoanOffer"
                  part="payload"/>
              </copy>
            </assign>
        </case>
        <otherwise>
            <assign>
                <copy>
                <from variable="loanOffer1" part="payload"/>
                <to variable="selectedLoanOffer"
                        part="payload"/>
                </copy>
            </assign>
        </otherwise>
    </switch>
   </scope>
   <invoke name="replyOutput"
          partnerLink="client"
          portType="tns:LoanFlowCallback"
          operation="onResult"
          inputVariable="selectedLoanOffer"/>
  </sequence>
</process>
```

ANNEXE E : SPÉCIFICATION BPEL DU PROCESSUS DECVALMACHINE

Cette annexe représente la spécification *BPEL* du processus `decValMachine` présenté au chapitre 6. Cet exemple a été extrait de l'article [25].

```
<process name="decValMachine"
         targetNamespace="http://acm.org/samples"
         suppressJoinFailure="yes"
         xmlns:tns="http://acm.org/samples"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-
         process/"
         xmlns:bpelx="http://schemas.oracle.com/bpel/extension"
         xmlns:ora="http://schemas.oracle.com/xpath/extension"
         xmlns:cx="http://schemas.collaxa.com/xpath/extension"
         xmlns:xsd=http://www.w3.org/2001/XMLSchema
         xmlns:ns0="http://schemas.xmlsoap.org/sl/">

 <partnerLinks>
       <partnerLink name="client"
                    partnerLinkType="tns:incDecLT"
                    myRole="incDecService"
                    partnerRole="incDecServiceRequester"/>
 </partnerLinks>

 <variables>
       <variable name="x" messageType="tns:x_nat"/>
       <variable name="dummy" messageType="tns:dummy"/>
       <variable name="r" messageType="tns:x_nat"/>
       <variable name="s" messageType="tns:state"/>
       <variable name="decErr" messageType="tns:invalid_dec"/>
 </variables>

 <sequence name="main">
     <receive name="receiveInput" partnerLink="client"
                     portType="tns:incDecPT" operation="initiate"
                     variable="x" createInstance="yes"/>
     <scope name="Machine">
          <sequence>
             <assign>
               <copy>
                  <from variable="x" part="idpart"
                        query="/ns0:x_natElement/ns0:x">
                  </from>
                  <to variable="s" part="idpart"
                      query="/ns0:stateElement/ns0:v"/>
               </copy>
             </assign>
             <assign>
                <copy>
```

```
    <from expression="false"/>
    <to variable="s" part="idpart"
        query="/ns0:stateElement/ns0:terminated"/>
  </copy>
</assign>
<while condition="not(bpws:getVariableData('s',
      'idpart','/ns0:stateElement/ns0: terminated') = 'true')">
<scope>
   <pick>
      <onMessage partnerLink="client" portType="tns:incDecPT"
                 operation="inc" variable="x">
        <assign>
         <copy>
          <from expression="bpws:getVariableData('s','idpart',
               '/ns0:stateElement/ns0:v') +
               bpws: getVariableData('x','idpart','/ns0:
               x_natElement/ns0:x')">
          </from>
         <to variable="s" part="idpart"
            query="/ns0:stateElement/ns0:v"/>
         </copy>
        </assign>
      </onMessage>
      <onMessage partnerLink="client" portType="tns:incDecPT"
                 operation="dec" variable="x">
          <switch>
           <case condition="bpws: getVariableData('s',
                'idpart', '/ns0:stateElement/ns0:v')
                &gt;= bpws: getVariableData('x','idpart',
                '/ns0: x_natElement/ ns0:x')">
                <assign>
                   <copy>
                    <from expression="bpws:getVariableData
                         ('s','idpart','/ns0: stateElement/
                          ns0:v') -  bpws: getVariableData
                          ('x','idpart',x_natElement/ns0:x')">
                    </from>
                   <to variable="s" part="idpart"
                      query="/ns0: stateElement/ ns0:v"/>
                </copy>
           </assign>
      </case>
      <otherwise>
        <throw faultName="invalid_dec" faultVariable="decErr"/>
      </otherwise>
   </switch>
 </onMessage>
 <onMessage partnerLink="client"  portType="tns:incDecPT"
           operation="val" variable="dummy">
           <assign>
             <copy>
               <from expression="true">
               </from>
                <to variable="s" part="idpart"
                     query="/ns0: stateElement/ ns0:terminated"/>
              </copy>
              <copy>
                 <from variable="s" part="idpart"
```

```
                                    query="/ns0: stateElement/ ns0:v">
                                </from>
                                 <to variable="r" part="idpart"
                                     query="/ns0:x_natElement/ ns0:x"/>
                            </copy>
                        </assign>
                    </onMessage>
                </pick>
            </scope>
          </while>
         </sequence>
        </scope>
        <invoke name="callbackClient" partnerLink="client"
                portType="tns:incDecPTCallback" operation="end"
                inputVariable="r"/>
        </sequence>
    </process>
```

ANNEXE F : MODÈLE PROMELA DU PROCESSUS LOANFLOW

```
/* fichier Promela généré à partir du */
/*        LoanFlow.bpel */

#define MAX  10
#define CHAINE_VIDE  0

mtype = {onResult, initiate, process};

typedef type_chaineCaracteres {
  int ch;
}
typedef type_LoanServiceResultMessage {
  int  payload;
}
typedef type_CreditRatingServiceFaultMessage {
  type_chaineCaracteres message;
}
typedef type_CreditRatingServiceResponseMessage {
  type_chaineCaracteres  payload;
}
typedef type_LoanFlowRequestMessage {
  type_chaineCaracteres  payload;
}
typedef type_LoanServiceRequestMessage {
  type_chaineCaracteres  payload;
}
typedef type_LoanFlowResultMessage {
  int  payload;
}
typedef type_CreditRatingServiceRequestMessage {
  type_chaineCaracteres  payload;
}


chan canal_client_In = [MAX] of {mtype, type_LoanFlowRequestMessage};
chan canal_UnitedLoanService_Out = [MAX] of
               {mtype, type_LoanServiceRequestMessage};
chan canal_StarLoanService_Out = [MAX] of
               {mtype, type_LoanServiceRequestMessage};
chan canal_creditRatingService_In = [MAX] of
               {mtype, type_CreditRatingServiceResponseMessage};
chan canal_StarLoanService_In = [MAX] of
               {mtype, type_LoanServiceResultMessage};
chan canal_creditRatingService_Out = [MAX] of
                {mtype, type_CreditRatingServiceRequestMessage};
chan canal_UnitedLoanService_In = [MAX] of
                 {mtype, type_LoanServiceResultMessage};
chan canal_client_Out = [MAX] of
                 {mtype, type_LoanFlowResultMessage};
```

```
type_LoanFlowRequestMessage input ;
type_CreditRatingServiceRequestMessage crInput ;
type_CreditRatingServiceResponseMessage crOutput ;
type_CreditRatingServiceFaultMessage crError ;
type_LoanServiceRequestMessage loanApplication ;
type_LoanServiceResultMessage loanOffer1 ;
type_LoanServiceResultMessage loanOffer2 ;
type_LoanFlowResultMessage selectedLoanOffer ;
bool terminate_signal = false ;
bool process_1_termine = false;
bool process_2_termine = false;


proctype process_1() provided (!terminate_signal)
{
  xs canal_UnitedLoanService_Out ;
  xr canal_UnitedLoanService_In ;

  canal_UnitedLoanService_Out ! initiate, loanApplication  ;
  canal_UnitedLoanService_In? onResult, loanOffer1  ;
  process_1_termine = true ;
}

proctype process_2() provided (!terminate_signal)
{
  xs canal_StarLoanService_Out ;
  xr canal_StarLoanService_In ;

  canal_StarLoanService_Out ! initiate, loanApplication  ;
  canal_StarLoanService_In? onResult, loanOffer2  ;
  process_2_termine = true ;
}

proctype LoanFlow() provided (!terminate_signal)
{
  xr canal_client_In ;
  xr canal_creditRatingService_In ;
  xs canal_creditRatingService_Out ;
  xs canal_client_Out ;


  canal_client_In? initiate, input  ;
  crInput.payload.ch = input.payload.ch ;
  canal_creditRatingService_Out ! process, crInput  ;
  canal_creditRatingService_In ? process, crOutput  ;
  input.payload.ch = crOutput.payload.ch ;
  loanApplication.payload.ch = input.payload.ch ;
  run process_1();
  run process_2();
  process_1_termine && process_2_termine ;
  if
  :: loanOffer1.payload > loanOffer2.payload ->
       selectedLoanOffer.payload = loanOffer2.payload ;
  :: else ->
      selectedLoanOffer.payload = loanOffer1.payload ;
  fi;
```

```
    canal_client_Out ! onResult, selectedLoanOffer  ;
}

proctype client() provided (!terminate_signal)
{
  xs canal_client_In ;
  xr canal_client_Out ;

  canal_client_In ! initiate, input  ;
  canal_client_Out ? onResult, selectedLoanOffer  ;
}

proctype UnitedLoanService() provided (!terminate_signal)
{
  xr canal_UnitedLoanService_Out ;
  xs canal_UnitedLoanService_In ;

  canal_UnitedLoanService_Out ? initiate, loanApplication  ;
  if
  :: skip -> loanOffer1.payload = 10 ;
  :: skip -> loanOffer1.payload = 0 ;
  fi ;
  canal_UnitedLoanService_In ! onResult, loanOffer1  ;
}

proctype StarLoanService() provided (!terminate_signal)
{
  xr canal_StarLoanService_Out ;
  xs canal_StarLoanService_In ;

  canal_StarLoanService_Out ? initiate, loanApplication  ;
  if
  :: skip -> loanOffer2.payload = 10 ;
  :: skip -> loanOffer2.payload = 0 ;
  fi ;
  canal_StarLoanService_In ! onResult, loanOffer2  ;
}

proctype creditRatingService() provided (!terminate_signal)
{
  xr canal_creditRatingService_Out  ;
  xs canal_creditRatingService_In ;

  canal_creditRatingService_Out ? process, crInput  ;
  canal_creditRatingService_In ! process, crOutput ;
}

init {
  run LoanFlow();
  run client () ;
  run UnitedLoanService() ;
  run StarLoanService () ;
  run creditRatingService () ;
}
```

ANNEXE G : MODÈLE PROMELA DU PROCESSUS DECVALMACHINE

```
/* fichier Promela généré à partir du */
/*        decValMachine.bpel */

#define MAX  10
#define CHAINE_VIDE  0

mtype = {val, initiate, dec, end, inc};

typedef type_chaineCaracteres {
  int ch;
}
typedef type_dummy {
  type_chaineCaracteres message;
}
typedef type_x_nat {
  int  idpart;
}
typedef type_state {
  bool  idpart;
}
typedef type_invalid_dec {
  type_chaineCaracteres message;
}

chan canal_client_Out = [MAX] of {mtype, type_x_nat};
chan canal_client_In = [MAX] of {mtype, type_x_nat, type_dummy};

type_x_nat x ;
type_dummy dummy ;
type_x_nat r ;
type_state s ;
type_invalid_dec decErr ;
type_x_nat   varFactice1;
type_dummy  varFactice2;
bool terminate_signal = false ;

proctype decValMachine() provided (!terminate_signal) {

  xr canal_client_In;
  xs canal_client_Out;

  canal_client_In? initiate, x , _  ;
  s.idpart = x.idpart ;
  s.idpart = false ;
  do
  ::s.idpart == false ->
      if
       :: canal_client_In ? inc, x , _  ->
              s.idpart = s.idpart + x.idpart ;
```

```
        :: canal_client_In ? dec, x , _  ->
                if
                :: s.idpart >= x.idpart ->
                     s.idpart = s.idpart - x.idpart ;
                :: else ->
                   skip;
                fi;
        :: canal_client_In ? val, _ , dummy  ->
                s.idpart = true ;
                r.idpart = s.idpart ;
        fi;
    :: else -> break ;
   od;
   canal_client_Out ! end, r  ;

}

proctype client()
{
   xs canal_client_In;
   xr canal_client_Out;

   canal_client_In ! initiate, x , _ ;
   do
    ::  canal_client_In ! inc, x , _;
    ::  canal_client_In ! dec, x , _ ;
    ::  canal_client_In ! val, _ , dummy ; break ;
    od;
    canal_client_Out ? end, r  ;
}

init {
   run decValMachine();
   run client();
}
```

BIBLIOGRAPHIE

1 T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic et S. Weerawarana. 2003. « Business Process Execution Language for Web Services ». 2nd release, Version 1.1.

2 F.v. Breugel et M. Koshkina. 2006. « Models and Verification of BPEL ». Technical Report, York University, Canada.

3 A. Martens. 2003. « On Compatibility of Web Services ». Petri Net Newsletter, vol. 65 (octobre), pages: 12-20.

4 A. Martens. 2005. « Consistency between Executable and Abstract Processes ». Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service, pages: 60-67, Hong Kong. IEEE.

5 C. Peltz. 2003. « Web Service Orchestration and Choreography A Look at WSCI and BPEL4WS ». Web Services Journal, vol. 3, no. 7 (juillet), pages: 30-35.

6 E. Christensen, F. Curbera, G. Meredith et S. Weerawarana. Site de W3C, [en ligne]. http://www.w3.org/TR/wsdl (page consultée le: 19 Octobre 2006)

7 W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, N. Russell, H.M.W. Verbeek et P. Wohed. 2005. « Life After BPEL? » in M. Bravetti et al. (Eds), Proc. of 2nd Int. Workshop on Web Services and Formal Methods, LNCS, vol. 3670, pages: 35-50.

8 E. Clarke, O. Grumberg et D. Peled. 1999. « Model Checking ». Cambridge, Mass.: MIT Press.

9 Ouvrage collectif - Coordination Philippe Schenoeblen. 1999. « Vérification de logiciels, techniques et outils du model-checking ». Paris : Vuibert.

10 G.J. Holzmann. 1992. « Protocol Design: Redifining the State of the Art ».IEEE Software, Network Protocols. Pages: 17-22.

11 S. Merz. 2001. « Model Checking: A Tutorial Overview ». LNCS 2067, pages: 3-38.

12 X. Fu, T. Bultan et J. Su. 2004. « Analysis of Interacting BPEL Web Services ». In Proceedings of the 13th International World Wide Web Conference (WWW), pages: 621-630, New York, NY.

13 X. Fu, T. Bultan, et J. Su. 2004. « WSAT: A Tool for Formal Analysis of Web Services ». In Proceedings of the 16th International Conference on Computer Aided Verification (CAV), pages: 501-504, Boston Massachusetts.

14 X. Fu, T. Bultan et J. Su. 2003. « Conversation Protocols : A Formalism for Specification and Verification of Reactive Electronic Services » In Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA), LNCS 2759, pages: 188-200, Springer, Santa Barbara.

15 X. Fu, T. Bultan, R. Hull et J. Su. 2003. « Conversation Specification: A New Approach to Design and Analysis of E-Service Composition » In Proceedings of the 12th International World Wide Web Conference (WWW), pages: 403-410. Budapest, Hungary.

16 S. Nakajima. 2002. « Model-checking Verification for Reliable Web Service ». In OOPSLA Workshop on Object-Oriented Web Services, pages 152–163.

17 S. Nakajima. 2005. « Lightweight Formal Analysis of Web Service Flows ». Progress in Informatics, vol. 1, no. 2 (novembre), pages: 57-76.

18 S. Nakajima. 2005. « Model-checking Behavioral Specification of BPEL Applications ». In Proceedings of the International Workshop on Web Languages and Formal Methods, vol. 151, no. 2 (juillet) of Electronic Notes in Theoretical Computer Science, pages: 89-105, New castle, UK. Elsevier.

19 S. Nakajima. 2002. « Verification of Web Service Flows with Model-Checking Techniques ». Proceedings of the 1st International Symposium on Cyber Worlds, pages: 378-386, Tokyo, Japan. IEEE.

20 S. Nakajima. 2004. « Model-Checking of Safety and Security Aspects in Web Service Flows ». N. Koch, P. Fraternali, et M. Wirsing, editors, Proceedings of the 4th International Conference of Web Engineering, LNCS 3140, pages: 488-501, Munich, Germany. Springer-Verlag.

21 J.A. Fisteus, L. S. Fernández et C.D. Kloos. 2005. « Applying Model Checking to BPEL4WS Business Collaborations ». Proceedings of the 2005 ACM symposium on Applied computing, pages: 826-830, Santa Fe, New Mexico.

22 H. Foster, S. Uchitel, J. Magee et J. Kramer. 2003. « Model-based Verification of Web Service Compositions ». 18th IEEE International Conference on Automated Software Engineering. Montreal, QC, Canada: IEEE Computer Society Press, pages: 152-161.

23 H. Foster. 2006. « A Rigorous Approach To Engineering Web Service Compositions ». PhD thesis, Imperial College London.

24 G. Tremblay. 2003. « Une introduction à la vérification de modèles ». Université du Québec à Montréal. Séminaire du doctorat en informatique cognitive.

25 G. Tremblay et J. Chae. 2005. « Towards Specifying Contracts and Protocols for Web Services ». H. Mili and F. Khendek, editors, Proceedings of the MCeTech Montreal Conference on eTechnologies, pages: 73-85, Montreal, Canada.

26 J.V.D. BOS et C. Laffra. 1989. « PROCOL: A Parallel Object Language with Protocols ». Conference proceedings on Object-oriented programming systems, languages and applications, pages: 95-102, New Orleans, Louisiana, United States.

27 S. Frølund, K. Govindarajan. 2003. « cl: A Language for Formally Defining Web Services Interactions ». Technical Report HPL-2003-208 (Hewlett-Packard).

28 G.J .Holzmann. 2003. « The Spin Model Checker: Primer and Reference Manual ». Addison-Wesley Professional.

29 G.J. Holzmann. 2000. « Using Spin ». Plan 9 Programmer Manual Documents, pages: 353-382, Vita Nuova Holdings Ltd, York England. 2nd edition.

30 G. J. Holzmann. 2005. « Software Model Checking with Spin ». Advances in Computers, vol. 65 (juillet), Ed. M. Zelkowitz, Elsevier Publ., Amsterdam, pages: 77-108.

31 G.J. Holzmann. 1997. « The Model Checker Spin ». IEEE Trans. on Software Engineering, vol. 23, no. 5 (mai), pages: 279-295.

32 S. Andler. 1979. « Predicate Path Expression ». In Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, pages: 226-236, San Antonio, Texas.

33 K. Salomaa et S. Yu. 1999. « Synchronization Expressions and Languages ». Journal of Universal Computer Science vol. 5, no. 9, pages: 610-621.

34 J.P. Queille et J. Sifakis. 1983. « Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness ». Acta Informatica, vol. 19, no. 3, pages: 195-220.

33 J.G. Cleaveland. 2001. « Program Generators with XML and Java ». Prentice Hall PTR.

35 D. Thomas et A. Hunt. 2001. « Programming Ruby: The Pragmatic Programmer's Guide ». Boston: Addison-Wesley.

36 M.B. Juric, B. Mathew et P. Sarang. 2006. « Business Process Execution Language for Web Services Second Edition ». Packt Publishing.

37 J. Chauvet. 2002. « Services Web avec SOAP, WSDL, UDDI, ebXML ». Eyrolles.

38 R.S. Pressman. 2005. « Software Engineering: A Practitioner's Approach ». Boston : McGraw-Hill Higher Education.

39 A. Salah, G. Tremblay et A. Chami. Janvier 2008. « Behavioral Interface Conformance Checking for WS-BPEL Processes ». International MCETECH Conference on e-Technologies, pages 253-256, IEEE Computer Society, Montréal, Canada.