An XSLT transformation from WS-CDL Specification to Promela Process

ReDCAD Laboratory, University of Sfax, Tunisia

1 Background

XSLT (Extensible Stylesheet Language: Transformations) is a W3C standard, it provides a powerful and flexible language to transform an XML document to one or more documents (XML, HTML, etc).
The principle function of XSLT is: An XSLT style contains rules that describe transformations (as is shown in 1). These rules are applied to a source XML

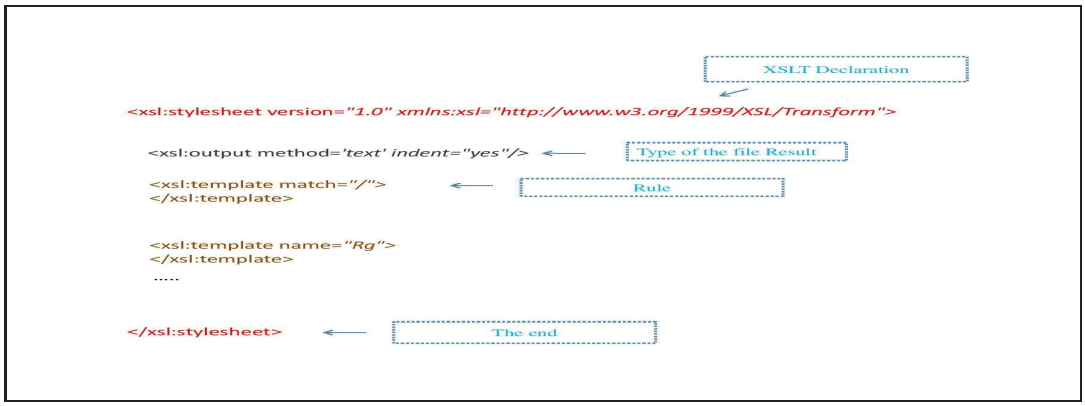

Figure 1: XSLT General Structure

document (WS-CDL in our case) to obtain a new result document (Promela code). The transformation is performed by a program called XSLT processor that is responsible for carrying out the instructions given in the stylesheet.

2 Mapping

To ensure reliable processing based grammars, we focus in this transformation to keep the same semantics as well as the same functionality of WS-CDL activities in Promela code.

2.1 General Rules

this transformation will affect different parts of WS-CDL. Indeed, the most important activity in any WS-CDL specification is <interaction activity>. For each one, we find:

- Participate with relationshipType, fromRoleTypeRef and toRoleTypeRef

As well, we find also:

- channelVariable;
- exchange which contains name, informationType and action.

We present successively the various transformtion stages. To begin we expose the rules adopted to promote the <proctype> transformation, and then we will show the different necessary changes that will affect all parts of the WS-CDL specification.

- **Proctype Transformation**:
  In each WS-CDL specification, raltion between two roles is defined throughout <RelationType> which is already in <Participate>. In fact, the former serves to identify the associated process to each interaction. This latter will be initiated by sending a message from process <FromRoleTypeRef>. This message will be received subsequently by the process <ToRoleTypeRef>. In our transformmation process, the name of the proctype will be the value of one of the process. Thereby, we will have two proctypes, one for messages transmission and the other for reception.

  Taking out the example of the proctype <fromRoleTypeRef>, it is essential, as the first step to begin the transformation, to recover the names of all <fromRoleTypeRef> process and give each of them a primary key in order to avoid duplications. Simultaneously, it is necessary to remove all special characters as well as specific symbols by replacing them with acceptable characters in Promela. Thus, the name of proctype will bee obtained by concatenating all names retrieved.

  Listing 1: Proctypes General Structure

```
proctype name_fromRoleTypeRef() {...}
proctype name_toRoleTypeRef() {...}
```

  It should be noted that the same rules will be applied respectively in the case of the transformation of <ToRoleTypeRef>.

Afterwards, the istructions of each proctype must be defined. In fact, each message will be under this form:

channelVariable[! | ?]exchange name,informationType/channelType

So, we will give, in order, the different transformations applied to all message components.

- **ChannelVariable Transformaion**:
  ChannelVariable is known as the channel of communication between the participants in WS-CDL. Since Promela tolerate the definition and use of proctypes, then it will be possible to define all communication channels outside proctypes, which will facilitate their use later in these latter. Then, and to complete this transformation, recovery of all existing channelVariable is needed while browsing the WS-CDL specificaion. Thereafter, each channel name will be associated with a primary key that will be used in the future to use these channels in the Promela code. In fact, duplication will be a source of error in the Promela code.

Listing 2: ChannelVariable Transformation

```
<xsl:key name="KeyChannel" match="//interaction" use="↩
    @channelVariable"/>
<xsl:for-eachselect="//interaction[generate-id() = generate-id(↩
    key('KeyChannel', @channelVariable)[1])]">
<xsl:text>
chan</xsl:text>
<xsl:choose>
<xsl:when test="contains(@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(@channelVariable, $↩
    FaultCara, $TrueCara),5,100)"/>
...
</xsl:when>
<xsl:otherwise>
<xsl:value-ofselect="translate(@channelVariable, $FaultCara, $↩
    TrueCara)"/>
...
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
```

Listing 3: Mtype General Structure

```
mtype { NAME_EXCHANGE};
mtype {informationType};
mtype {channelType};
```

It is well to note that the channel will contain two messages. So, each of <ExchangeName>, <InformationType> and <ChannelType> will be defined in a separate mtype and out of this sub proctype in Promela.
In fact this will facilitate their calls in the various proctype.

- **ExchangeName Transformation**:

For example, if we take the case of the name of exchange, we will go all the WS-CDL file and retrieve all names that exist in exchange. As well,we will allocate to each name found a primary key to avoid the redundancy name. As previously, the specific characters will be replaced by a suitable character in Promela; and progressively we will put the names recovered in uppercase. It should be noted that the same rules will be applied to obtain the <InformationType> and <ChannelType> transformation.

- **Activities Transformation**:
  As well, in WS-CDL interaction, the attribute <action> can have only <repond> or <request>.

  In fact, the latter means that there must have an answer to toRoleTypeRef. While the former means that there is an application that was made âĂŃâĂŃby fromRoleTypeRef.

  In Promela, the symbol "!" and "?" indicate that there are, respectively, a demande and a repsonse. Based on these definitions, a channel (channelvariable) can apply for a message:
  channelVariable! exchangename, informationType/ ChannelType
  As well, a channel may have a receipt message:
  channelVariable? exchangename, informationType / ChannelType.

  Over and above, to apply the transformation rules on the various instructions , it is imperative to distinguish between the proctypes of <fromRoleTypedef> and those of <toRoleTypedef.>

  After defining and transforming proctypes with instruction, it will be imperative to define the activities [1] that may exist. Taking the case of the structural activity "sequence", it means we will execute in the same order interactions obtained from the WS-CDL file. So we will just put the instructions transformed in the same order as the WS-CDL.

2.2 Sequence

Listing 4: Sequence Example in WS-CDL

```
.
.
.
<sequence>
<interaction name="Buyer requests a Quote - this is the initiator" ↩
    operation="requestForQuote" channelVariable="Buyer2SellerC" ↩
    initiate="true">
<participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" ↩
    toRole="SellerRoleType"/>
<exchange name="request" informationType="RequestForQuoteType" action↩
    ="request">
<send/>
<receive/>
</exchange>
<exchange name="response" informationType="QuoteType" action="respond↩
    ">
<send/>
```

```
<receive/>
</exchange>
</interaction>
</sequence>
.
.
.
```

By applying the following XSLT rules (Listing 5), we will have the Promela code as is shown in Listing 6.

Listing 5: XSLT Rule

```
<xsl:for-each select="exchange">
<xsl:if test="@action='request'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" /> !
<xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
    ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
    TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 ;
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
    TrueCara)" /> !
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
    ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@informationType,$FaultCara, $TrueCara↩
    )" />
 </xsl:otherwise>
 </xsl:choose>
 ;
</xsl:otherwise>
</xsl:choose>
</xsl:if>


<xsl:if test="@action='respond'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" /> ?
<xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
    ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
```

```
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
     TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 ;
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
     TrueCara)" />   ?
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
     ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
     $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@informationType,$FaultCara, $TrueCara↩
    )" />
 </xsl:otherwise>
 </xsl:choose>
 ;
</xsl:otherwise>
</xsl:choose>
</xsl:if>
```

Listing 6: Promela Code: Sequence Structure

```
mtype {RequestForQuoteType,QuoteType};
mtype {REQUEST,RESPONSE};
mtype {n};
chan Buyer_SellerC = [n] of {mtype, mtype};

proctype BuyerRoleTypeSellerRoleType() {
Buyer_SellerC ! REQUEST, RequestForQuoteType ;
 Buyer_SellerC ? RESPONSE, QuoteType ;}

proctype SellerRoleTypeBuyerRoleType() {
Buyer_SellerC ? REQUEST, RequestForQuoteType ;
Buyer_SellerC ! RESPONSE, QuoteType ; }

init { atomic {
run BuyerRoleTypeSellerRoleType() ;
run SellerRoleTypeBuyerRoleType () ;
}}
```

2.3 Choice

The <choice> activity implies that there is a choice between two or more instructions in a given condition. In WS-CDL specification, the choice instruction is under this form:

Listing 7: Choice Structure in WS-CDL specification

```
.
.
.
<choice>
<sequence>
<interaction name="Credit Checker fails credit check" operation="↩
    creditFailed" channelVariable="Seller2CreditChkC">
<participate relationshipType="SellerCreditCheck" fromRole="↩
    SellerRoleType" toRole="CreditCheckerRoleType" />
<exchange name="creditCheckFails" informationType="CreditRejectType" ↩
    action="respond">
<send />
<receive />
</exchange>
```

```
</interaction>
</sequence>
<sequence>
<interaction name="Credit Checker passes credit" operation="creditOk" ↩
    channelVariable="Seller2CreditChkC">
<participate relationshipType="SellerCreditCheck" fromRole="↩
    BuyerRoleType" toRole="CreditCheckerRoleType" />
<exchange name="creditCheckPasses" informationType="CreditAcceptType" ↩
    action="respond">
<send />
<receive />
</exchange>
</interaction>
<interaction name="Shipper sends delivery details to buyer" operation=↩
    "deliveryDetails" channelVariable="DeliveryDetailsC">
<description type="description">Pass back shipping details to the ↩
    buyer</description>
<participate relationshipType="ShipperBuyer" fromRole="ShipperRoleType↩
    " toRole="BuyerRoleType" />
<exchange name="sendDeliveryDetails" informationType="↩
    DeliveryDetailsType" action="request">
<send />
<receive />
</exchange>
</interaction>
</sequence>
</choice>
.
.
.
```

The transformation will be initiated by the existence of the tag <choice> will be replaced by "if". Thereafter, the interaction will be transformed. The end of the selection will be indicated by "fi". The adopted XSLT rule is the following:

Listing 8: XSLT Rule for choice transformation in fromRoleTypeRef proctype

```
<xsl:for-each select="//interaction/participate[generate-id() = ↩
    generate-id(key('KeyFromRole', @fromRole|@fromRoleTypeRef)[1])]">
<xsl:choose>
<xsl:when test="contains(@fromRole|@fromRoleTypeRef,'tns:')">
<xsl:value-of select="substring(translate(@fromRole|@fromRoleTypeRef, ↩
    $FaultCara, $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@fromRole|@fromRoleTypeRef, $FaultCara↩
    , $TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 </xsl:for-each>(){
<xsl:for-each select="choice">
<xsl:when test="sequence/interaction|interaction">
if
<xsl:for-each select="sequence|interaction">
::
<xsl:for-each select="interaction/exchange|exchange">
<xsl:if test="@action='request'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" />  !
<xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
    ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
```

```
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
    TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 ;
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
    TrueCara)" />   !
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
     ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@informationType,$FaultCara, $TrueCara↩
    )" />
 </xsl:otherwise>
 </xsl:choose>
 ;
</xsl:otherwise>
</xsl:choose>
</xsl:if>


<xsl:if test="@action='respond'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" />  ?
<xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
     ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
    TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 ;
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
    TrueCara)" />   ?
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
     ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@informationType,$FaultCara, $TrueCara↩
    )" />
 </xsl:otherwise>
 </xsl:choose>
 ;
</xsl:otherwise>
</xsl:choose>
</xsl:if>
</xsl:for-each">
fi;
```

```
} init { atomic { run
<xsl:for-each select="//interaction/participate[generate-id() = ↩
    generate-id(key('KeyFromRole', @fromRole|@fromRoleTypeRef)[1])]">
<xsl:choose>
<xsl:when test="contains(@fromRole|@fromRoleTypeRef,'tns:')">
<xsl:value-of select="substring(translate(@fromRole|@fromRoleTypeRef, ↩
    $FaultCara, $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@fromRole|@fromRoleTypeRef, $FaultCara↩
    , $TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 </xsl:for-each>
 (); run
<xsl:for-each select="//interaction/participate[generate-id() = ↩
    generate-id(key('KeyToRole', @toRole|@toRoleTypeRef)[1])]">
<xsl:choose>
<xsl:when test="contains(@toRole|@toRoleTypeRef,'tns:')">
  <xsl:value-of select="substring(translate(@toRole|@toRoleTypeRef, $↩
      FaultCara, $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@toRole|@toRoleTypeRef, $FaultCara, $↩
    TrueCara)" />
</xsl:otherwise>
</xsl:choose>
 </xsl:for-each> ();
}
```

The following code is relatif to the result of choice transformation in fromRoleTypeRef proctype:

Listing 9: Promela code for choice activity in fromRoleTypeRef proctype

```
mtype { CreditRejectType, CreditAcceptType, DeliveryDetailsType };
mtype { CREDITCHECKFAILS, CREDITCHECKPASSES, SENDDELIVERYDETAILS };
mtype {n};
chan Seller_CreditChkC = [n] of {mtype, mtype};
chan DeliveryDetailsC = [n] of {mtype, mtype};

proctype SellerRoleTypeBuyerRoleTypeShipperRoleType () {
if
::
 Seller_CreditChkC ? CREDITCHECKFAILS   , CreditRejectType ;
  ::
Seller_CreditChkC ? CREDITCHECKPASSES   , CreditAcceptType ;
 DeliveryDetailsC ! SENDDELIVERYDETAILS   , DeliveryDetailsType;
fi;
}
proctype CreditCheckerRoleTypeBuyerRoleType() {
if
::
 Seller_CreditChkC ! CREDITCHECKFAILS   , CreditRejectType ;
  ::
Seller_CreditChkC ! CREDITCHECKPASSES   , CreditAcceptType ;
 DeliveryDetailsC ? SENDDELIVERYDETAILS   , DeliveryDetailsType;
fi;
}

init { atomic {
run BuyerRoleTypeSellerRoleType() ;
run CreditCheckerRoleTypeBuyerRoleType() ;
}}
```

2.4 Parallel

Listing 10: Parallel Structure in WS-CDL specification

```
.
.
.
<parallel>
<sequence>
<interaction name="Buyer accepts the quote and engages in the act of ↩
    buying" operation="quoteAccept" channelVariable="Buyer2SellerC">
<participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" ↩
    toRole="SellerRoleType" />
<exchange name="Accept Quote" informationType="QuoteAcceptType" action↩
    ="request">
<send />
<receive />
</exchange>
</interaction>
</sequence>
<sequence>
<interaction name="Buyer send channel to seller to enable callback ↩
    behavior" operation="sendChannel" channelVariable="Buyer2SellerC">
<participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" ↩
    toRole="SellerRoleType" />
<exchange name="sendChannel" channelType="2BuyerChannelType" action="↩
    request">
<send />
<receive />
</exchange>
</interaction>
</sequence>
</parallel>
.
.
.
```

To transform <parallel>, each activity will be defined in a controlled subâĂŃâĂŃ-processes using a channel.

Listing 11: XLT Rule for Parallel transformation

```
<xsl:for-each select="//parallel">
<xsl:for-each select="sequence/interaction/exchange|interaction/↩
    exchange">
 proctype From_sub_Process
 <xsl:value-of select="position()" />  (chan i){
  <xsl:text />
<xsl:for-each select="exchange">
<xsl:if test="@action='request'">
 <xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
 <xsl:value-of select="substring(translate(../@channelVariable, $↩
     FaultCara, $TrueCara),5,100)" />   !
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
     ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
     $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
     TrueCara)" />
 </xsl:otherwise>
</xsl:choose>  ;
```

```
 </xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
     TrueCara)" />   !
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
       ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
     $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
     TrueCara)" />
 </xsl:otherwise>
 </xsl:choose>  ;
 </xsl:otherwise>
 </xsl:choose>
 </xsl:if>
<xsl:if test="@action='respond'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" />   ?
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
       ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
 <xsl:value-of select="substring(translate(@informationType,$FaultCara↩
     , $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
     TrueCara)" />
  </xsl:otherwise>
  </xsl:choose>  ;
  </xsl:when>
<xsl:otherwise>
  <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
      TrueCara)" />   ?
  <xsl:value-of select="translate(@name, $minuscules, $majuscules)" />↩
        ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
     $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
     TrueCara)" />
  </xsl:otherwise>
  </xsl:choose>  ;
  </xsl:otherwise>
  </xsl:choose>
  </xsl:if>
  </xsl:for-each>
<xsl:for-each select="//interaction[generate-id() = generate-id(key('↩
    KeyChannel', @channelVariable)[1])]">
  i !
<xsl:choose>
<xsl:when test="contains(@channelVariable,'tns:')">
 <xsl:value-of select="substring(translate(@channelVariable, $↩
     FaultCara, $TrueCara),5,100)" />
  ;
  </xsl:when>
<xsl:otherwise>
```

```
 <xsl:value-of select="translate(@channelVariable, $FaultCara, $↩
     TrueCara)" />   ;
  </xsl:otherwise>
  </xsl:choose>
  </xsl:for-each>
  }
  </xsl:for-each>
</xsl:for-each>
```

Thereafter, these sub-processes are executed within the main process. To illustrate our proposal, we assume that we have <parallel> with two activities. So let's define two sub-processes (for FromRole and ToRole) then we will run in the main process in the following manner:

Listing 12: Promela Code for Parallel structure

```
mtype { QuoteAcceptType};
mtype {_BuyerChannelType};
mtype { ACCEPTQUOTE, SENDCHANNEL };

mtype {n};
chan Buyer_SellerC = [n] of {mtype, mtype};

proctype FromSubProcess1() {
Buyer_SellerC ? ACCEPTQUOTE , QuoteAcceptType ;
}
proctype FromSubProcess2() {
Buyer_SellerC ? SENDCHANNEL , _BuyerChannelType ;
}
proctype ToSubProcess1() {
Buyer_SellerC ! ACCEPTQUOTE , QuoteAcceptType ;
}
proctype ToSubProcess2() {
Buyer_SellerC ! SENDCHANNEL , _BuyerChannelType ;
}

proctype FromRole_name() {
/*instruction*/

Chan syncro = [2] of {int};
run FromSubProcess1(syncro);
run FromSubProcess2(syncro);
do ::
full(syncro) -> break;
od;

/*instruction*/
}

proctype ToRole_name() {
/*instruction*/

Chan syncro = [2] of {int};
run ToSubProcess1(syncro);
run ToSubProcess2(syncro);
do ::
full(syncro) -> break;
od;

/*instruction*/
}

init { atomic {
run FromRole_name();
run ToRole_name() ();
}}
```

2.5 Workunit

In WS-CDL, workunit can be written in two ways.

- It is a simple repetition of the condition.
- It is repetition until a stop condition.

Listing 13: Workunit structure in WS-CDL example

```
.
.
.
<workunit name="Repeat until bartering has been completed" repeat="↩
    barteringDone = false">
<sequence>
<interaction name="Buyer accepts the quote and engages in the act of ↩
    buying" operation="quoteAccept" channelVariable="Buyer2SellerC">
<participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" ↩
    toRole="SellerRoleType" />
<exchange name="AcceptQuote" informationType="QuoteAcceptType" action=↩
    "request">
<send />
<receive />
</exchange>
</interaction>
<assign roleType="BuyerRoleType">
<copy name="copy">
<source expression="true" />
<target variable="cdl:getVariable('barteringDone','','')" />
</copy>
</assign>
</sequence>
<sequence>
<interaction name="Buyer updates the Quote - in effect requesting a ↩
    new price" operation="quoteUpdate" channelVariable="Buyer2SellerC"↩
    >
<participate relationshipType="BuyerSeller" fromRole="BuyerRoleType" ↩
    toRole="SellerRoleType" />
<exchange name="updateQuote" informationType="QuoteUpdateType" action=↩
    "request">
<send />
<receive />
</exchange>
</interaction>
</sequence>
</workunit>
```

Listing 14: XSLT Rule for Workunit transformation

```
<xsl:for-each select="//interaction/participate[generate-id() = ↩
    generate-id(key('KeyFromRole', @fromRole|@fromRoleTypeRef)[1])]">
<xsl:choose>
<xsl:when test="contains(@fromRole|@fromRoleTypeRef,'tns:')">
<xsl:value-of select="substring(translate(@fromRole|@fromRoleTypeRef, ↩
    $FaultCara, $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@fromRole|@fromRoleTypeRef, $FaultCara↩
    , $TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 </xsl:for-each>(){
<xsl:for-each select="workunit">
<xsl:when test="sequence/interaction|interaction">
```

```
do
<xsl:for-each select="sequence|interaction">
::
<xsl:for-each select="interaction/exchange|exchange">
<xsl:if test="@action='request'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" />  !
<xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
      ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
    TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 ;
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
    TrueCara)" />  !
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
      ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@informationType,$FaultCara, $TrueCara↩
    )" />
 </xsl:otherwise>
 </xsl:choose>
 ;
</xsl:otherwise>
</xsl:choose>
<xsl:choose>
 <xsl:when test="not(../assign) if :: condition = true ; break ; fi;</↩
    xsl:when>
<xsl:when test="assign[position()>0] and not(../sequence/choice)">↩
    break;</xsl:when>
</xsl:choose>
</xsl:if>


<xsl:if test="@action='respond'">
<xsl:text />
<xsl:choose>
<xsl:when test="contains(../@channelVariable,'tns:')">
<xsl:value-of select="substring(translate(../@channelVariable, $↩
    FaultCara, $TrueCara),5,100)" />  ?
<xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
      ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
    $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
 <xsl:value-of select="translate(@informationType,$FaultCara, $↩
    TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 ;
```

```
 <xsl:value-of select="translate(../@channelVariable, $FaultCara, $↩
     TrueCara)" />   ?
 <xsl:value-of select="translate(@name, $minuscules, $majuscules)" /> ↩
       ,
<xsl:choose>
<xsl:when test="contains(@informationType,'tns:')">
<xsl:value-of select="substring(translate(@informationType,$FaultCara,↩
     $TrueCara),5,100)" />
 </xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@informationType,$FaultCara, $TrueCara↩
    )" />
 </xsl:otherwise>
 </xsl:choose>
 ;
</xsl:otherwise>
</xsl:choose>
<xsl:choose>
 <xsl:when test="not(../assign) if :: condition = true ; break ; fi;</↩
     xsl:when>
<xsl:when test="assign[position()>0] and not(../sequence/choice)">↩
     break;</xsl:when>
</xsl:choose>
</xsl:if>
</xsl:for-each">
od;
} init { atomic { run
<xsl:for-each select="//interaction/participate[generate-id() = ↩
     generate-id(key('KeyFromRole', @fromRole|@fromRoleTypeRef)[1])]">
<xsl:choose>
<xsl:when test="contains(@fromRole|@fromRoleTypeRef,'tns:')">
<xsl:value-of select="substring(translate(@fromRole|@fromRoleTypeRef, ↩
     $FaultCara, $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@fromRole|@fromRoleTypeRef, $FaultCara↩
     , $TrueCara)" />
 </xsl:otherwise>
</xsl:choose>
 </xsl:for-each>
 (); run
<xsl:for-each select="//interaction/participate[generate-id() = ↩
     generate-id(key('KeyToRole', @toRole|@toRoleTypeRef)[1])]">
<xsl:choose>
<xsl:when test="contains(@toRole|@toRoleTypeRef,'tns:')">
  <xsl:value-of select="substring(translate(@toRole|@toRoleTypeRef, $↩
       FaultCara, $TrueCara),5,100)" />
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="translate(@toRole|@toRoleTypeRef, $FaultCara, $↩
     TrueCara)" />
</xsl:otherwise>
</xsl:choose>
 </xsl:for-each> ();
}
```

Listing 15: Promela code for Workunit

```
mtype { QuoteAcceptType, QuoteUpdateType };
mtype { ACCEPTQUOTE, UPDATEQUOTE };
mtype {n};
chan Buyer_SellerC = [n] of {mtype, mtype};

proctype BuyerRoleTypeSellerRoleType() {
do
::
 Buyer_SellerC ? ACCEPTQUOTE , QuoteAcceptType ; break;
  ::
```

```
Buyer_SellerC? UPDATEQUOTE, QuoteUpdateType;
;
od;
}
proctype BuyerRoleTypeSellerRoleType() {
do
::
 Buyer_SellerC ! ACCEPTQUOTE , QuoteAcceptType ; break;
  ::
Buyer_SellerC! UPDATEQUOTE, QuoteUpdateType;
;
od;
}

init { atomic {
run BuyerRoleTypeSellerRoleType() ;
run BuyerRoleTypeSellerRoleType() ;
}}
```

3 Summary

Table 1: The different mappings between WS-CDL constructors and Promela code

WS-CDL construtors	Promela concept
participantType (<fromRole> and <toRole>)	Proctype Name
acivity	channelvariable[!|?]exchange name;informationType;channelvariable
choice	if...fi;
workunit	do...od;
choice	if...fi;
Choice in workunit without assign	do :: if :: ... if :: condition = true ; break ; fi; ... fi ; od ;
Choice in workunit with assign	do :: if :: .... Break ; fi ; od ;
Workunit without assign in choice	if :: do :: .... if :: condition = true ; break ; fi; .... od ; fi ;
Workunit with assign in choice	if :: do :: .... Break ; od ; fi ;
Parallel with sequence or choice or workunit (with and without assign)	Proctype subprocess (chan i) {All send activity "!" }

References

[1] S. Rebai, N. Guermouche, H. Hadj-Kacem, and A. Hadj-Kacem. Towards error-handling-aware choreography to orchestration transformation approach. *Int. J. Collaborative Enterprise*, 3(2/3):151–166, 2013.